6-1997

The Column Multiplicity Problem in Decomposition of Functions and Relations

Rahul Malvi
Portland State University

THESIS APPROVAL

The abstract and thesis of Rahul Malvi for the Master of Science in Electrical and Computer Engineering were presented June 2nd 1997, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]
Marek A. Perkowski, Chair

[redacted]
Malgorzata E. Chrzanowska-Jeske

[redacted]
Laszlo Csanky
Representative of the Office
of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]
Rolf Schaumann, Chair
Department of Electrical Engineering

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on Aug. 15, 1997

ABSTRACT

An abstract of the thesis of Rahul Malvi for the Master of Science in Electrical and Computer Engineering presented June 2nd 1997.

Title: The Column Multiplicity Problem in Decomposition of Functions and Relations

Finding the column multiplicity in Functional Decomposition has been known to be one of the most important problems to be solved in the process of functional decomposition of discrete functions. A lot of research has been done in this field with many new heuristics generated to find the column multiplicity, but there has not been an evaluation of the algorithms on the kinds of graphs that occur in decomposition and whether having an exact method to calculate the column multiplicity is useful from the overall design goals. The intent of this thesis was to investigate the column multiplicity problem, in order to find out how different heuristic algorithms compare with an exact algorithm for finding the column multiplicity, and to find out if an exact graph coloring is actually required or not.

In order to investigate this problem of column multiplicity in functional decomposition, two graph coloring programs, and a multi-coloring program were written: one of the graph coloring programs is an exact graph coloring, the other graph coloring program and the multi-coloring program are both based on heuristic algorithms. The two graph coloring programs have been compared in this thesis on randomly generated graphs. These programs were incorporated into the multi-valued decomposer of functions and relations *MVGUD* which was developed at Portland State University. Extensive testing of *MVGUD* with these graph coloring and multi-coloring programs has been done in this thesis. *MVGUD* was tested on both circuit benchmarks and on machine learning benchmarks.

THE COLUMN MULTIPLICITY PROBLEM IN DECOMPOSITION OF FUNCTIONS AND RELATIONS

by

RAHUL MALVI

A thesis submitted for the partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1997

ACKNOWLEDGEMENTS

I would like to thank Dr. Marek A. Perkowski, my advisor, who introduced me to the areas of Logic Synthesis, and Decomposition of Functions and Relations. This thesis would not have been possible without the guidance and encouragement I received from Dr. Perkowski. I would also like to thank Dr. Malgorzata E. Chrzanowska-Jeske and Dr Laszlo Csanky for their valuable comments which helped to improve this thesis.

I would also like to thank my parents for their support and encouragement, and I would like to thank Uju for her constant support and encouragement. Also, I would like to thank Shirley, Laura and Ellen for all their assistance and support.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Functional Decomposition is one of the most general methods to solve engineering and science optimization problems. Functional Decomposition is an NP-complete problem. Because of its low time efficiency it is rarely used in industry, while Algebraic Factorization and Binary Decision Diagrams are used instead. Recently though, there has been some strong renewed interest in using Functional Decomposition for applications like FPGA mapping and PLA Decomposition. Functional Decomposition involves breaking down a function of larger complexity into smaller blocks of less complexity and then these smaller blocks can be broken down themselves. The Functional Decomposition group at Portland State University was doing this research for Wright Patterson Air Force base and for Abtech Corporation. The Pattern Theory group (PTG) at the Avionics Laboratory of Wright Laboratories, at Wright Patterson Air Force Base develops new system-level concepts for military applications, mostly based on Machine Learning and Image Processing Technologies. Machine Learning is making the machine learn(induce concept descriptions) as much as possible with as little data as possible. Hence the idea is that the machine must train on the set of samples and find the pattern in the set of samples. Then the machine can apply the pattern to all the data and thus recreate the values of the original samples not given to it. Most, if not all, functions taken from real-life applications have been found to have some pattern. There are many methods which can be used in Machine Learning, like, Neural Nets, Fuzzy Logic, Decision Trees, Binary Decision Diagrams(BDDs), Functional Decision Diagrams(FDDs), Kronecker Decision Diagrams(KDDs), Sum of Products(SOP), e.t.c. All of these methods are

computationally expensive, but usually less than the Curtis Decomposition. The approach of the PTG is based on Logic Synthesis methods: predominantly the Curtis Decomposition of Boolean functions [16]. The new approach of the PTG will allow both automatic learning of any kind of images, and automatic creation of algorithms, from examples of their behavior. The PTG has developed the programming system FLASH [7], which uses many Decomposition ideas, and is a Testbed for machine learning based on the logic synthesis approach. The applications considered by Wright Laboratories include Pattern Recognition, Algorithm Creation, Data Compression, Machine Learning and Logic Minimization. Functional Decomposition is the approach used by the group at Portland State University. Functional Decomposition can be applied, to minimize a function by finding a pattern in it, and thus breaking the original function into smaller functions. The structure of the decomposed blocks constitutes the algorithm of the function and thus the decomposition finds the pattern in the function. These smaller functions or sub-functions are the "features" or "concepts" found in the original function

Over the last twenty years a lot of effort has gone into the many synthesis methods like BDDs, SOP e.t.c. and it has been found that these methods achieve good results 80% of the time and are fast, while the Ashenhurst [1] or the Curtis [2] Decompositions will achieve good results nearly 100% of the time but are very slow. The difficulty of applying the Ashenhurst/Curtis Decomposition is that, as the number of input variables and terms increases, the Decomposition time increases exponentially. Hence we must be able to reduce the computational time to an acceptable level for a given number of variables(10 to 30 variables), while sacrificing a very small amount of solution complexity(2% to 3%). In many industrial and military applications it has been found that the data typically consist of a very large number of don't cares, typically more than 99%. Hence the goal of this research was to develop a Decomposer capable of decomposing functions with many input variables, but with a very large percentage of don't

cares. Also, in this research we wanted to use different algorithms or techniques such as Graph Coloring, Maximum Cliques, Compatibility and Incompatibility Graphs, both exact and approximate, and test the various heuristics and exact methods in conjunction with the Ashenhurst/Curtis Decomposition method in order to decompose problems much more efficiently and without losing much in accuracy.

Functional Decomposition involves three basic steps namely Variable Partitioning, calculating Column Multiplicity and Encoding. A lot of research has been done in Functional Decomposition, but it is still not known how important each step is and whether one step is more important than the other. Two Functional Decomposers were developed at Portland State University: one a multi-valued Decomposer *MVGUD* [26] and the other a binary Decomposer *GUD* [26]. *GUD* stands for General Universal Decomposer. *GUD* was in turn put into a larger program called *Multis* which also had two other Functional Decomposers: *TRADE* [5, 6] and *DEMAIN* [4] incorporated in it. *Multis* is capable of using different Decomposers at different steps of the Decomposition thus making it possible to combine different strategies, while decomposing a function.

This thesis deals with the Column Multiplicity problem in Functional Decomposition. There are basically four methods to solve the Column Multiplicity Problem in Functional Decomposition: Set Covering, Graph Coloring, Clique Partitioning and Clique Covering. All these are NP-complete problems. Chapter 2 of this thesis gives an introduction to Decomposition of Boolean Functions. Chapter 3 presents two new Graph Coloring programs: one a Heuristic method, and the other an Exact method. Also in Chapter 3 ideas for two new exact Graph Coloring programs are presented. In Chapter 4 the heuristic Graph Coloring method is compared with the exact Graph Coloring method on graphs generated randomly, in order to see how these two programs compare in terms of their computational time, and how close to exact are the solutions generated by the heuristic Graph Coloring program. The two programs have been incorporated into the Functional

Decomposer *MVGUD*, and this decomposer already has a Greedy Clique Partitioning incorporated into it. So in Chapter 5 all three methods are compared, by testing them on functions generated from real life problems, to see how well these programs solve the Column Multiplicity problem. The testing was done on two types of Benchmarks: *MCNC* benchmarks, generated from Circuit minimization, and Machine Learning Benchmarks from areas of Pattern Recognition, Machine Learning and Knowledge Discovery in Databases. In Chapter 6 the concept of decomposition of relations is introduced, and in this Chapter a new heuristic Multi-Coloring program is presented. In Chapter 7 this Multi-Coloring program is tested in Decomposition of Functions and Relations in order to see if the Multi-Coloring can solve the Column Multiplicity problem better than a Graph Coloring or a Clique Partitioning. Finally, Chapter 8 summarizes the work and addresses my point of view of what the future tasks of the Functional Decomposition group at Portland State University should be.

CHAPTER 2

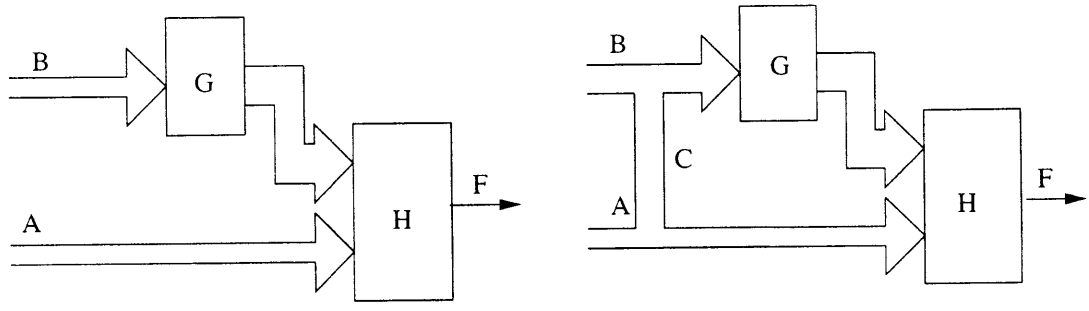
GENERAL DECOMPOSITION OF BOOLEAN FUNCTIONS

2.1 Introduction

Decomposition is a method to break down a large function into a number of smaller functions, which are easier to analyze and realize than the large input function. Boolean Decomposition uses a Boolean representation. Many different kinds of Decompositions have been developed. The first was the Ashenhurst Decomposition defined in [1], then Curtis [2] extended the Ashenhurst Decomposition, making it more general. This thesis deals with the Curtis decomposition, and in this Chapter, the Curtis Decomposition will be presented.

2.2 Generalized Functional Decomposition

The Boolean Decomposition involves breaking a function $F(A, B)$ into $H(g(B)$-$, A)$ where the number of inputs of H is smaller than that of F. If $A \bigcap B = \emptyset$ then it is called a Disjoint Decomposition, as shown in Figure 2.1(a), else it is



(a) A Disjoint Decomposition

(b) A Non Disjoint Decomposition

Figure 2.1: Difference between Disjoint and Non Disjoint Decompositions

ab \ cd	00	01	11	10
00	X	0	0	1
01	X	X	0	1
11	0	0	X	0
10	0	1	1	0

(a) A Karnaugh Map

ab \ cd	00	01	10	11
00	X	0	1	0
01	X	X	1	0
10	0	1	0	1
11	0	0	0	X

(b) A Decomposition Chart

Figure 2.2: A Kmap vs a Decomposition Chart

called a Non-Disjoint Decomposition, as shown in Figure 2.1(b). In Figure 2.1(b) $C \subset A$.

2.3 Basic Notions and Definitions

2.3.1 A Karnaugh Map vs a Decomposition Chart

Figure 2.2(a) shows a Karnaugh map representation of a function having four inputs and one output, Figure 2.2(b) shows the same function represented in a Decomposition chart. The only difference between a Karnaugh map and a Decomposition chart is that the row and column indices in a Karnaugh map are in Gray code while they are in binary order in a Decomposition chart. So from the Karnaugh map it can be seen that for an input variable combination of 1101 the output is "0".

2.3.2 A Minterm

Definition 2.1 *For a function* **F** *with* **n** *inputs, a minterm is a combination of values 0,1 of all its* **n** *argument values.*

Row	a	b	c	y
1	1	0	1	1
2	1	-	1	1
3	-	-	1	1
4	1	0	0	0

Table 2.1: Table showing Minterms and Cubes

In the Table 2.1, row 1 is a minterm as none of the inputs are don't cares. In a Karnaugh map of a function, each cell in the Karnaugh map represents a minterm of the function.

2.3.3 A Cube

Definition 2.2 *For a function* **F** *with* **n** *inputs, a cube is any combination of* **n** *in which, at least one of the inputs in* **n** *is a don't care.*

In the Table 2.1, row 2 is a cube as input "b" is a don't care.

2.3.4 Hamming Distance

Definition 2.3 *The Hamming distance of two minterms or cubes is the number of bits the two minterms or cubes differ in.*

In Table 2.1 the minterm in row 4 and the minterm in row 1 have a Hamming distance of one, since they differ only in the value of input "c". In Table 2.1 the cube in row 2 and the minterm in row 4, also have a Hamming distance of one, because they have the same value for input "a", and for input "b" row 2 has a don't care which means it can take any value, hence they only differ in the value of input variable "c".
Any two minterms, or any two cubes can be combined together in a Karnaugh map if they have a Hamming distance of one.

2.3.5 The Cofactor of a Function

Definition 2.4 *Any Boolean function can be represented by an expansion* $f_a = af_a + \bar{a}f_{\bar{a}}$. *This is called a Shannon Expansion. Here* $f_a = f\mid_{a=1}$ *is called the positive cofactor of function* f *with respect to variable* a, *and* $f_{\bar{a}} = f\mid_{a=0}$ *is called the negative cofactor of function* f *with respect to variable* a.

Cofactors can be easily seen from a Karnaugh map, in Figure 2.2(a) column "00" represents the cofactor of the input function $f_{\bar{c},\bar{d}} = f\mid_{c=0,d=0}$, and row "11" represents the cofactor of the input function $f_{a,b} = f\mid_{a=1,b=1}$. Hence the columns represent the cofactors with respect to the variables "c" and "d", and the rows represent the cofactors with respect to the variables "a" and "b" in the Karnaugh map shown in Figure 2.2(a).

2.3.6 The Free Set and the Bound Set

As one of the first steps in Decomposition we split the input variables into two parts, the free set A and the bound set B.

Definition 2.5 *In a Disjoint Decomposition,* $F(A,B) = H(g(B),A)$ *the free set(A) is the set of variables forming the rows of the Decomposition chart, and the bound set(B) is the set of variables forming the columns of the Decomposition chart. In a Disjoint Decomposition* $A \bigcap B = \emptyset$. *In a Non Disjoint Decomposition* $A \bigcap B \neq \emptyset$.

In Figure 2.2(b) variables a, b are in the Free Set, and variables c, d are in the Bound Set.

2.3.7 Compatible and Incompatible Columns of a Decomposition Chart

Definition 2.6 *Two columns of a Decomposition chart are said to be compatible if they are the same or if assigning values to don't cares can make them the same. Compatibility between two columns is an incomplete tautology check between the two columns.*

In Figure 2.2(b) column 00 and column 10 are compatible. Column 00 and column 01 are not compatible because minterm (1000) is a "0" while minterm (1001) is a "1", hence the two columns are incompatible.

2.3.8 Column Multiplicity

Definition 2.7 *The Column Multiplicity(v) of a Function is the number of incompatible columns in the Decomposition chart representing the function.*

In Figure 2.2(b) column 00 is compatible with column 10 and columns 01 and 11 are compatible, hence the column multiplicity is "2".

2.3.9 A Compatibility graph vs an Incompatibility graph

Figure 2.3 shows the columns of a Karnaugh map represented, as an Incompatibility Graph, in Figure 2.3(b), and as a Compatibility Graph in Figure 2.3(c). Nodes represent columns in the Karnaugh map labeled as 1,2,3,4 in Figure 2.3(a).

Definition 2.8 *In an incompatibility graph an edge between two nodes, where the nodes represent columns of a Karnaugh map, means that the two columns are incompatible. In a compatibility graph an edge between two nodes means that the two columns are compatible.*

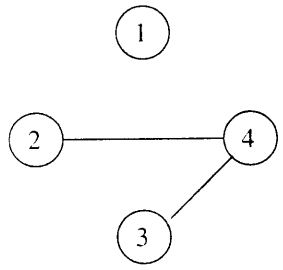
In the Karnaugh map shown in Figure 2.3(a), because column 1 has all don't cares, it has no edges in the incompatibility graph shown in Figure 2.3(b), and edges with all the other nodes in the compatibility graph shown in Figure 2.3(c).

2.3.10 Ashenhurst and Curtis Decompositions

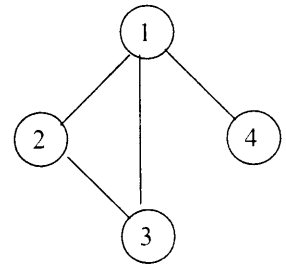
The two main Decompositions that will be discussed here are Ashenhurst [1] and Curtis Decomposition [2]. Figure 2.4 shows an Ashenhurst Decomposition. The basic requirement for an Ashenhurst Decomposition is that the G block can have at most one output. Disjoint Ashenhurst Decompositions occur rarely for completely specified binary functions. Figure 2.5 shows a Curtis Decomposition,

ab \ cd	1 (00)	2 (01)	3 (11)	4 (10)
00	X	0	0	1
01	X	X	0	1
11	X	0	X	0
10	X	1	1	0

(a) A Karnaugh Map of Function F



(b) Incompatibility Graph of Function F



(c) Compatibility Graph of Function F

Figure 2.3: A Compatibility graph and an Incompatibility graph of a Function

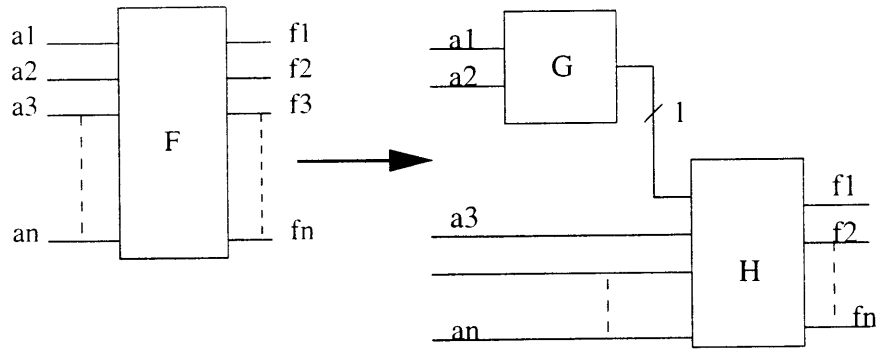


Figure 2.4: An Ashenhurst Decomposition

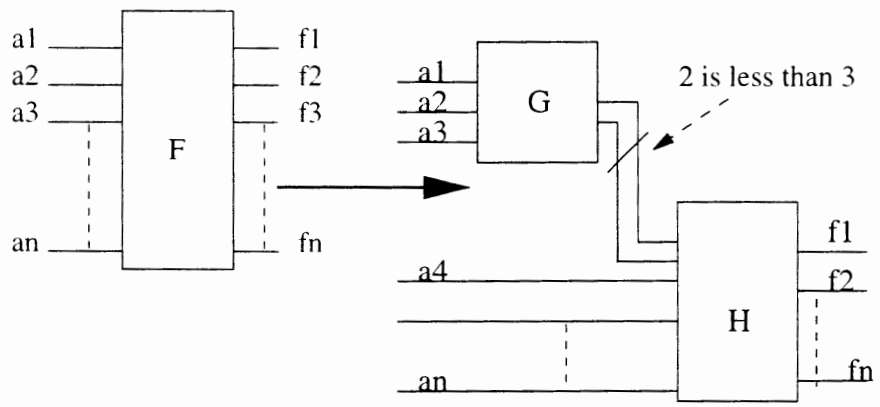


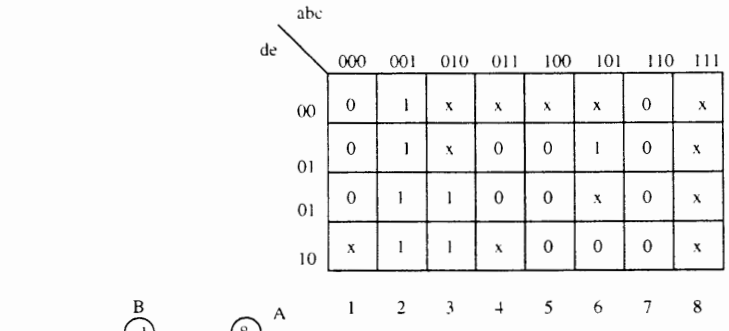
Figure 2.5: A Curtis Decomposition

Curtis Decomposition occurs very often. For a Curtis Decomposition to exist, the block G must have less outputs than inputs.

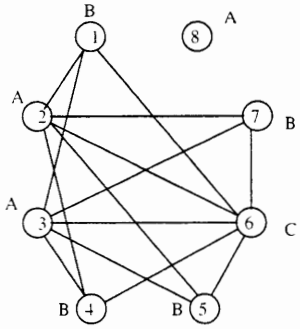
2.4 The Steps involved in performing a Curtis Decomposition

Now the steps involved in performing a Curtis Decomposition will be explained with the help of an example.

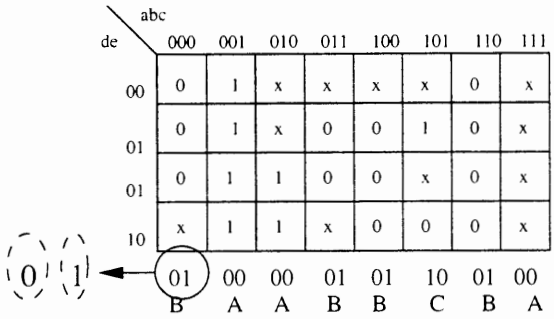
Figure 2.6 shows how the Curtis Decomposition process is performed on a binary function. The aim here is to decompose the function F into two functions which we call "G" and "H". The inputs to function "G" will be the variables in the bound set, and the inputs to the function "H" will be the free set variables and the outputs of the "G" function. As the first step to decomposition a bound set is selected. This is the Variable Partitioning Step. Figure 2.6(a) shows the Decomposition chart of a function F which has five inputs and one output. As can be seen from this Figure, the bound variables have been chosen as a, b and c, and the free variables are d and e. As the next step the columns which are compatible have to be grouped together. For this step either a Compatibility or an Incompatibility Graph can be created. In this example an Incompatibility Graph as shown in Figure 2.6(b) is created. The Incompatibility Graph is a direct mapping of the columns of the Decomposition chart, the number of nodes in the



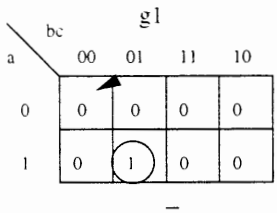
(**a**) Decomposition Chart of Function F



(**b**) Incompatibility Graph of Function F

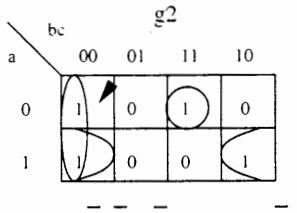


(**c**) Decomposition Chart after Graph Coloring showing Encoding



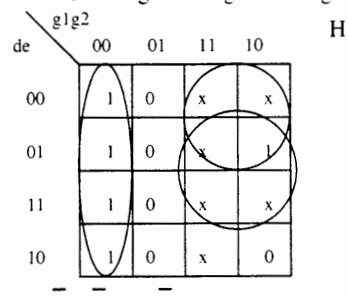
$g1 = a\,b\,c$

(**d**) Karnaugh map of g1



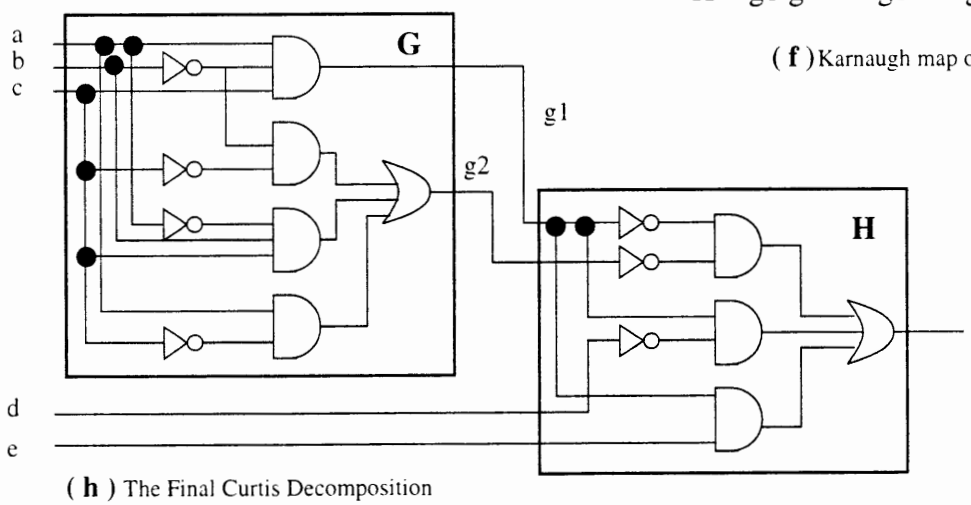
$g2 = b\,c + a\,b\,c + a\,c$

(**e**) Karnaugh map of g2



$H = g1\,g2 + d\,g1 + e\,g1$

(**f**) Karnaugh map of H



(**h**) The Final Curtis Decomposition

Figure 2.6: The Steps involved in Performing a Curtis Decomposition

Incompatibility Graph is equal to the number of columns in the Decomposition chart. The Graph Coloring technique is used to color the graph, and thus find out which columns are compatible. Graph coloring will be explained in more detail in Chapter 3. After assigning the colors it is found that there are three colors involved (colors are represented by letters in Figure 2.6(b)). Figure 2.6(b) shows the colored graph. Figure 2.6(c) shows the columns with the colors assigned to them.

Now the encoding step is done. Since two bits are required to encode the three colors (as found from the graph coloring), this tells us that there are two outputs from the G block. These are intermediate variables which we call g_1 and g_2. Figure 2.6(c) shows the encodings chosen here, in an arbitrary way. Columns with color A are given the code "00", columns with color B are given the code "01", and columns with color C are given the code "10". The Karnaugh maps representing the Functions g_1 and g_2 are shown in Figure 2.6(d) and Figure 2.6(e), respectively. We know that the inputs to the G Block are variables a, b and c, hence these variables form the indices of the Karnaugh maps of g_1 and g_2. In order to make the Karnaugh map of function g_1, for abc = "000", it is seen in Figure 2.6(c) that the encoding given to that column is "01", so the first bit, bit "0" is put in the Karnaugh map of function g_1 at location "000" in Figure 2.6(d), and the second bit, bit "1" is put in the Karnaugh map of function g_2 at location "000" in Figure 2.6(e). In this way the Karnaugh maps of functions g_1 and g_2 are created. The "H" function is shown in Figure 2.6(f). Variables "d" and "e" of the free set, and the outputs of the G block g_1 and g_2 form the indices of this Karnaugh map. For column with g_1,g_2 = "00" which is colored with A, all the columns in the decomposition chart shown in Figure 2.6(c) having color A are combined together to form one column which is included in all the columns with color A. In order to form this column don't cares can get assigned values. This finally results in a column with all four 1's, which forms column "00" in Figure 2.6(f). In this way the Karnaugh map of the H function is created. Then the Karnaugh maps of functions g_1,g_2 and H are solved and the logic required to

implement the G and H blocks is obtained. Figure 2.6(g) shows the final Curtis Decomposition with the logic required to implement the G and H blocks.

2.5 Applications of Functional Decomposition

Functional Decomposition is an NP-Complete Problem, and due to this complexity it is not very commonly used in the industry, though recently there has been a renewed interest in Decomposition. In TRADE [5] Functional Decomposition provides very good results for mapping a function into a Xilinx Field Programmable Gate Array(FPGA). Functional Decomposition can be used for Look up table (LUT) based FPGAs to minimize the number of LUTs used. Decomposition can also be used for VLSI design and other FPGAs such as Fine Grain FPGAs, Atmel, Motorola etc and for complex Programmable Logic Devices (PLD's). Decomposition can also be used in PLA Decomposition. Decomposition is also used in areas like image processing, machine learning, knowledge discovery, knowledge acquisition, database optimization, AI, image coding, automatic theorem proving and verification of software and hardware. In Machine learning, Decomposition can be used to extract the pattern in a function. This pattern is basically the reduced decomposed function. Decomposition of relations which will be introduced in Chapter 6, finds applications in areas like state assignment of non-deterministic state machines, Machine Learning and Knowledge Discovery from databases. Thus we conclude this Chapter by stating that Decomposition is a very powerful tool, but it still has to attain its full potential in the industry.

CHAPTER 3

SOME NEW APPROACHES TO GRAPH COLORING: TWO PROGRAMS AND TWO IDEAS

Finding the Column Multiplicity for a chosen bound set is very important for the success of a Functional Decomposer program, and a high percentage of the run time of the Functional Decomposer is spent on the Column Minimization part of Decomposition. What is needed is not only a fast method to solve the problem, but also one which produces results as close to optimal as possible.
Here we want to find out what is the role of Column Minimization in the overall success of a Decomposer; especially, in terms of the calculation time, the memory usage, and the quality of results. We want to investigate how the answers to these questions depend on the type of data, for instance on the percent of don't cares, or on the density of graphs in question. Presently the decomposer introduced by Pedram et al [17] achieves the best results; this program uses a set covering approach. But this decomposer is primarily for circuit applications. It is commonly believed that the primary success of this decomposer is due to the good data structures used. It is believed that reformulating the problem from a set covering approach to a graph coloring approach may significantly improve the efficiency of the decomposition. Graph Coloring for decomposition was introduced by Muzio and Wesselkamper [20] and Perkowski [13]. It is not a new idea in logic synthesis but is often overlooked as an alternative approach to Set Covering but is actually a more suitable choice in many problems. Recently there are more approaches that use Graph Coloring [21]
There are basically four methods to find the Column Multiplicity in Functional Decomposition, namely Set Covering, Graph Coloring, Clique Partitioning and

Clique Covering. A relation between the columns of the Karnaugh map of a function, for a given bound and free sets can be represented as a Compatibility Graph or as an Incompatibility Graph. If represented as a Compatibility Graph, nodes which are connected together are compatible nodes and can be colored with the same color. This is called Clique Covering. If the graph is represented as an Incompatibility Graph then nodes which do not have a common edge can be colored with the same color. This is called Graph Coloring. Even though there has been a lot of research done in the field of Graph Coloring and Functional Decomposition, no one, to our knowledge, has compared these methods, or evaluated the importance of finding minimal solutions to the problem of Column Multiplicity in the Ashenhurst/Curtis Decompositions.

In this Chapter two new approaches to Graph Coloring will be introduced. One of the Graph Coloring methods is a heuristic algorithm and the other is an Exact Graph Coloring. The heuristic Algorithm(*DOM*), which uses dominations to color the graph was presented in [18] by Perkowski. The idea for the Exact Graph Coloring Algorithm(*EXOC*) was a modification of an idea given to me by Perkowski. Also in this Chapter ideas for two new exact Graph Coloring algorithms are presented. The first idea extends the heuristic Graph Coloring algorithm to an exact Graph Coloring algorithm using dominations(*EXDOM*). The second idea combines the heuristic Graph Coloring algorithm and the exact Graph Coloring algorithm to an exact Graph Coloring algorithm. Before introducing the Graph Coloring algorithms and programs, some basic notions and definitions are presented.

3.1 Basic Notions and Definitions

3.1.1 Dominations in an Incompatibility Graph

Definition 3.1 *Some node "A" in an incompatibility graph dominates some other node "B" in the graph if the following is satisfied:*
1)Node "A" and node "B" have no common edge.

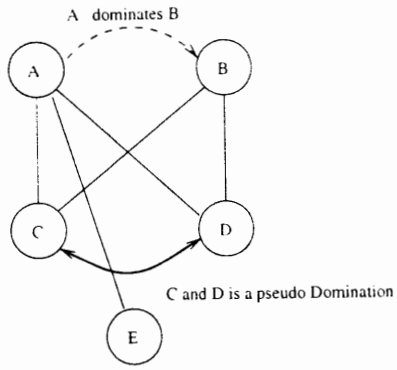


Figure 3.1: Dominations and Pseudo Dominations in an Incompatibility Graph

2)Node "A" has edges with all the nodes that node "B" has edges with.
3)Node "A" has at least one more edge than node "B".

In Figure 3.1, node "A" dominates node "B" since it satisfies the conditions for domination. When two nodes have a domination then both the nodes can be colored with the same color.

Definition 3.2 *If conditions 1) and 2) for dominations is satisfied and node "A" has the same number of edges as node "B", then it is called a pseudo domination.*

In Figure 3.1, nodes C and D, have a pseudo domination.

Theorem 3.1 *If any node "A" in a graph dominates any other node "B" in the graph, node "B" can be removed from the graph, and in a pseudo domination any one of the nodes "A" or "B" can be removed.*

Proof 3.1 *The proof for the Theorem 3.1 is that if any node "A" in a graph dominates any other node "B" in the graph, it means that node "B" is a proper subset of node "A" hence by removing node "B" we are not changing the number of colors in the graph, as node "A" holds all the edge information that node "B" holds.*

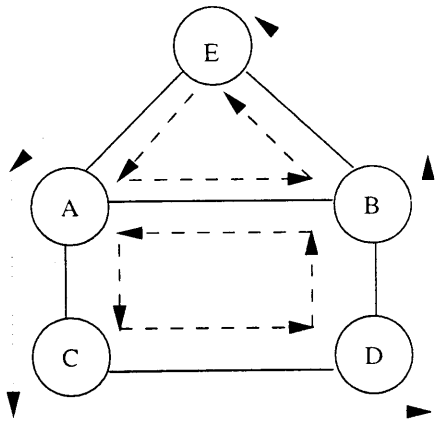


Figure 3.2: A Cyclic Graph

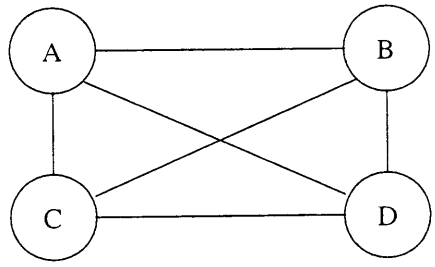


Figure 3.3: A Complete Graph

3.1.2 A Cyclic Graph and Cycles in an Incompatibility Graph

Definition 3.3 *In a cyclic graph no dominations can be found, but pseudo dominations may be found.*

Definition 3.4 *A Cycle in an Incompatibility Graph is a circuit that passes through every vertex exactly once.*

In Figure 3.2 circuit consisting of vertices E, A and B is a cycle.

3.1.3 A Complete Graph

Definition 3.5 *A Complete graph [15] is one in which all vertice pairs are connected.*

In a complete graph

$$total\_edges = \frac{nodes * (nodes - 1)}{2}$$

where *total_edges* is the sum of all the edges in the graph. In a Complete Graph no dominations or pseudo dominations can be found. In a Complete Graph all the nodes must have a unique color. A Complete Graph is a special case of a Cyclic Graph. Figure 3.3 shows an example of a Complete Graph.

3.2 A New Approach to a Heuristic Graph Coloring Using Dominations

In this section a new heuristic algorithm which uses the concept of dominations(explained in Section 3.1.1) to color an incompatibility graph is presented. This heuristic graph coloring algorithm has been given the name *DOM*, and in future sections it will be referred to by this name. In order to properly introduce *DOM*, first some examples which show how *DOM* colors an incompatibility graph are presented, and then the algorithm and implementation details of *DOM* are given.

3.2.1 An Example showing how *DOM* colors a non cyclic graph

The following steps explain how *DOM* colors a non cyclic graph.

1. Figure 3.4(a) shows an Incompatibility Graph. As can be seen Node 2 is dominated by Node 1, so in Figure 3.4(b) Node 2 is removed and it is remembered that it was dominated by Node 1. The proof that a dominated node can be removed from the incompatibility graph was given in Section 3.1.1.

2. Next, in Figure 3.4(b) Node 5 is removed as it is dominated by Node 4, and it is remembered that Node 4 dominates Node 5.

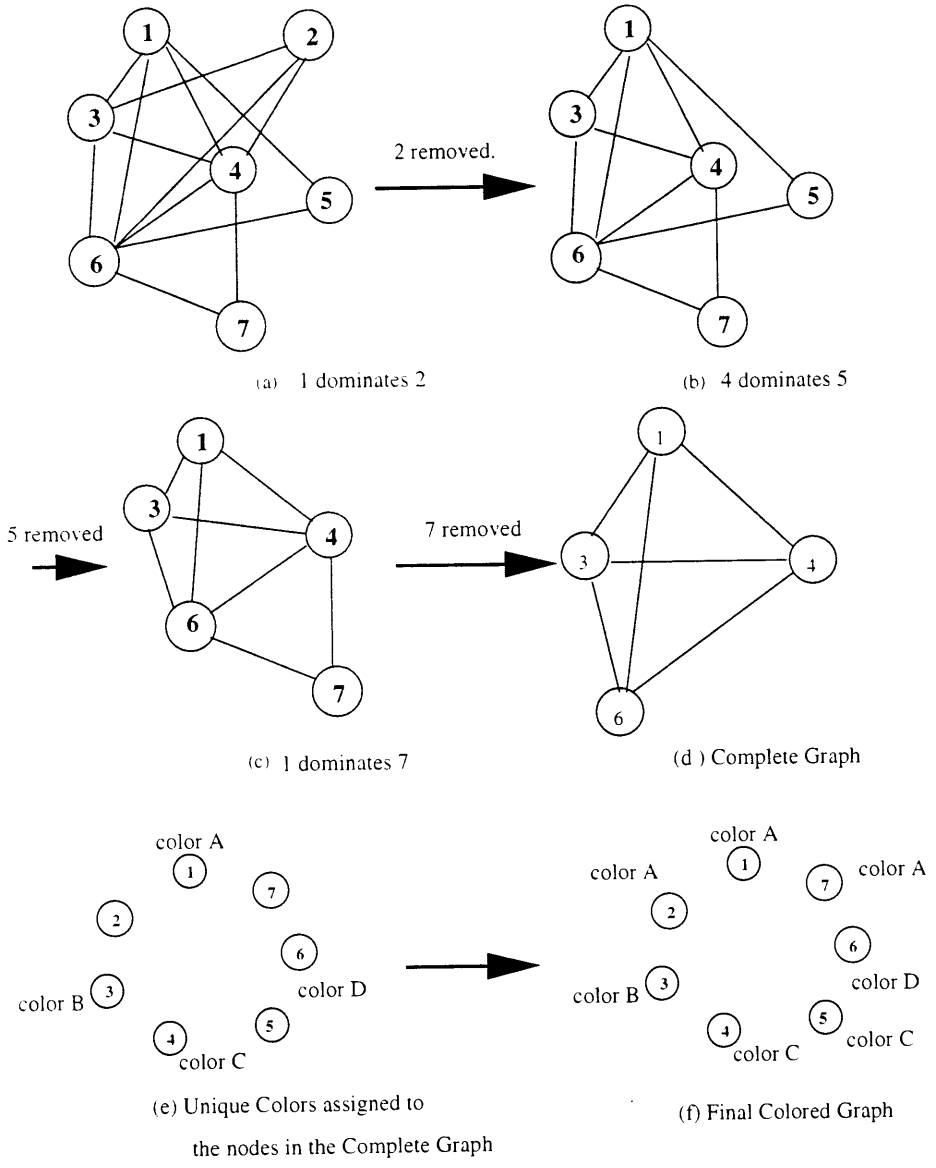


Figure 3.4: Example showing how *DOM* colors a non cyclic graph

3. Then in Figure 3.4(c), it can be seen that Node 7 is dominated by both nodes 1 and 3, the choice made is the first node which is Node 1, and Node 7 is removed. It is now remembered that Node 1 now dominates Node 2 and Node 7.

4. After removing Node 7 the resulting graph shown in Figure 3.4(d) is a complete graph, so go to Step 5. In a complete graph(shown in Section 3.1.3), each node is connected to all the other nodes, each node in the complete graph must have a unique color.

5. In Figure 3.4(e), each node in the Complete Graph is given a unique color.

6. Finally in the last step in Figure 3.4(f) the dominated nodes are colored with the same color as the dominating node. The color assignments are
   Color A {1, 2, 7}.
   Color B {3}.
   Color C {4, 5}.
   Color D {6 }.
   Thus in this way the graph is colored. Figure 3.4(e) shows the completely colored graph. Four colors were used which is the minimum required for this graph.

3.2.2 An Example showing how *DOM* colors a cyclic Graph

This Example illustrates how *DOM* colors a cyclic graph.

1. An incompatibility graph is shown in Figure 3.5(a), as can be seen this graph has a number of cycles.

2. As the first step the graph is checked for dominations, but no dominations are found in this graph, so the first node is removed from the graph, which is node 1, and it is assigned a minimum possible color which in this case is color A.

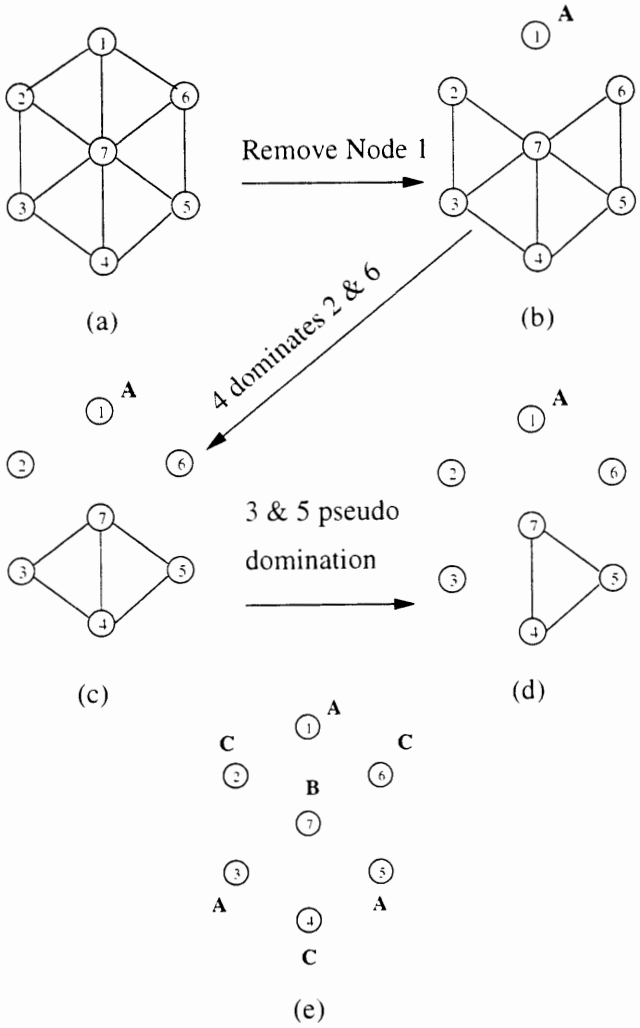


Figure 3.5: Example showing how *DOM* colors a cyclic graph

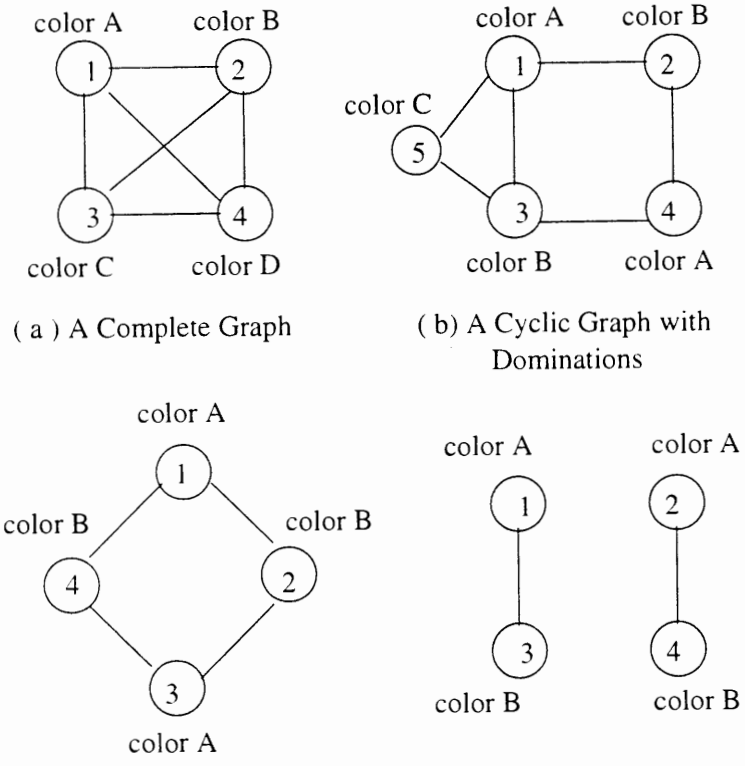
3. This results in a new graph, shown in Figure 3.5(b). In this graph node 4 dominates node 2 and node 6. So node 2 and node 6 are removed from the graph, and it is remembered that node 4 dominates node 2 and node 6.

4. On removing node 2 and node 6, in the resulting graph shown in Figure 3.5(c) nodes 3 and 5 have a pseudo domination so the first one of these nodes which is node 3 is removed, and then node 4, 5, and 7 form a complete graph. The complete graph is shown in Figure 3.5(d).

5. Now nodes are colored with the minimum possible color, and each dominated node is given the same color as the node which dominated it. The coloring is shown in Figure 3.5(e). Three colors were used to color the graph, which is the minimum required for this graph. The color assignments are:
   Color A {1, 3, 5}.
   Color B {7}.
   Color C {2, 4, 6}.

3.2.3 Different Graphs Possible in Decomposition, and how they are handled by *DOM*

Figure 3.6 shows some possible classes of graphs, that might be generated during the different stages of Functional Decomposition, and the colorings that *DOM* would find for these graphs. The heuristic assumption of our approximate algorithms will be that all graphs will fall into one of these classes, only they will have different number of nodes and different number of edges. Figure 3.6(a) shows a Complete Graph, in this case *DOM* will generate a unique color for each node in the Complete Graph. Figure 3.6(b) shows a cyclic graph with dominations, in this case *DOM* will find that node 3 dominates node 2, and node 1 dominates node 4. Then *DOM* will remove the dominated nodes 2 and 4, and the resultant graph is a complete graph. Then each node in the complete graph is given a unique color and the dominating nodes are colored the same color as the nodes which



(a) A Complete Graph

(b) A Cyclic Graph with Dominations

(c) A Cyclic Graph with Pseudo Dominations.

(d) A Disjoint Graph

Figure 3.6: The Different Classes of Incompatibility Graphs that may be generated in the steps of Functional Decomposition

dominate them. Figure 3.6(c) shows a cyclic graph with pseudo dominations, here *DOM* will find that nodes 1 and 3 have a pseudo domination and will give them the same color, and nodes 2 and 4 have a pseudo domination and will give them the same color. Figure 3.6(d) shows a disjoint graph, here *DOM* will choose one of the nodes, assign it a minimum possible color, and then remove it from the graph and go back to checking for dominations. As can be seen in all the cases shown, *DOM* will generate the minimum solution.

3.2.4 Implementation Details of *DOM*

The Algorithm for *DOM* has been divided into two parts:

1. Function *check_dominations* which checks for dominations in the graph, it has two possible return values.

   (i) Returns "TRUE" if the graph was complete.

   (ii) Returns "FALSE" if the graph was cyclic, or disjoint.

2. Function *graph_coloring* is the top level function, which calls *check_dominations*.

3.2.4.1 Pseudo Code for Function *graph_coloring*

```
graph_coloring( )
{
    /* Set initial status to FALSE */
    int status := FALSE;
    /* Continue to loop as long as check_dominations returns "FALSE" */
    while ( status == FALSE ) {
        status := check_dominations( ); /* Call function to check for
dominations*/
            if ( status == FALSE ) { /* If return value is False */
              /** Color the first node, choose minimum possible color
                for this node, and remove it from the graph. **/
```

```
                choose_node( ); /* Choose the first node */
                color_node( );  /* Color the first node */
            }
            else if ( status == TRUE ) /* Graph is complete */
                break;              /* Break out of the while loop */
    }
    /* On landing here color the nodes in the graph.
     if a node has dominations, color the dominated nodes
     the same color as the dominating node */
     color_nodes( );
     exit(0);       /* Exit from program */
}
```

3.2.4.2 Algorithm for Function *check_dominations()*

Step 1: Choose the node with the most incident edges, if tie choose the first one.

Step 2: If the graph is complete, then, return with value TRUE.

Step 3: Check for dominations of the chosen node with all the other nodes.

Step 4: If a domination is found, then, store the information of the dominating node, and the node that it dominates.

Step 5: Remove the edges of the dominated node from the incompatibility graph.

Step 6: If the chosen node has not been checked with all the other nodes go to Step 2, else go to Step 7.

Step 7: If all the nodes have been checked, return with value FALSE, else go to Step 1.

3.2.4.3 Pseudo Code for Function *check_dominations()*

```
check_dominations()
```

```
{
    int     dominating, dominated;
    /* Loop for the number of nodes */
    for ( i = 1; i ≤ number_of_nodes; i++ ) {
        if ( complete_graph( ) ) /* Function checks if graph is complete */
            return TRUE;
        dominating = select_a_node(); /* Choose a node */
        /* Loop for all the nodes - 1 */
        for ( dominated = 1; dominated ≤ number_of_nodes - 1; dominated++ ) {
            if ( domination_found(dominating, dominated) ) {
                /* Store the information of dominating and dominated nodes */
                mark_dominating_node( dominating, dominated );
                /* Remove the edges of the dominated node */
                remove_edges( dominated );
            }
        }
    }
    /* If we land here it means we were unable to find dominations */
    return FALSE;
}
```

3.2.5 The strong and weak points of *DOM*

By the examples shown in Figure 3.4 and Figure 3.5 we showed cases when *DOM* will generate the best solution. We do not claim that *DOM* will generate the best solution for all possible graphs, but we think it will perform well in most cases.

One weak point of the algorithm is when no dominations are found, then a node is selected and assigned a minimum possible color. If the coloring of this node is a bad choice(a bad choice is one which will lead to a non minimal coloring), it will result in a solution which is not minimal. Another weak point of the algorithm

is when it fails to find dominations or pseudo dominations at any stage of the coloring. In a graph like this, since *DOM* will choose a node and color it at each stage, it is basically doing a greedy coloring, and a greedy algorithm will get the same solution in a much faster time. But we think that these kind of complicated graphs(worst case graphs) will rarely, if ever, occur during the steps of Functional Decomposition, which is the application in which we are interested.

The strong point of *DOM* is that it can find the minimum solution without backtracking in all the cases when the graph which results after checking for dominations is a complete graph. Thus this program will be effective in finding the minimum solution in all non-cyclic graphs. The program may not be able to find the best solution in a cyclic graph.

3.3 An idea which extends *DOM* to an Exact Graph Coloring

In this Section an idea of how the algorithm for *DOM* can be made into an algorithm, which will always generate the minimum number of colors for an incompatibility graph is presented. This new algorithm will be called *EXDOM*. In the previous section it was shown that *DOM* will always generate the exact solution whenever the resulting graph at any stage of checking for dominations is complete, it is only when the graph is cyclic that *DOM* may not generate the exact solution. On coming across such a graph, a tree search is done in order to make *DOM* exact. This tree search is explained with the help of an example.

The tree search method shown in Figure 3.7 will be used by *EXDOM* whenever the function *check_dominations* (this algorithm was explained in Section 3.2.4.3) which checks for dominations in the graph has a return value of FALSE. Figure 3.7 shows how *EXDOM* colors a cyclic graph.

1. The block labeled SNODE 1 in Figure 3.7 is a cyclic graph which has 6 nodes in the cycle (*EXDOM* has no knowledge about the nature of the cyclic nodes, that is how many cycles are there). So the information of which nodes are in the cycle, is stored.
   Select node 1. Assign the minimum possible color to node 1 (The minimum possible color is the first possible color that can be assigned to a node which does not conflict with any nodes already colored).
   Color chosen is A. Remove the edges of node 1.

2. Removing Node 1 results in a new graph, shown in block labeled SNODE2 a). In this graph the function *check_dominations* returns a value of TRUE, since it finds a complete graph. The steps of checking for dominations are shown in block labeled SNODE 2 in Figure 3.7. So in SNODE 2 a solution is obtained, which uses 3 colors to color the graph. The color assignment is:
   Color A: {1, 3, 5},
   Color B: {2, 4},

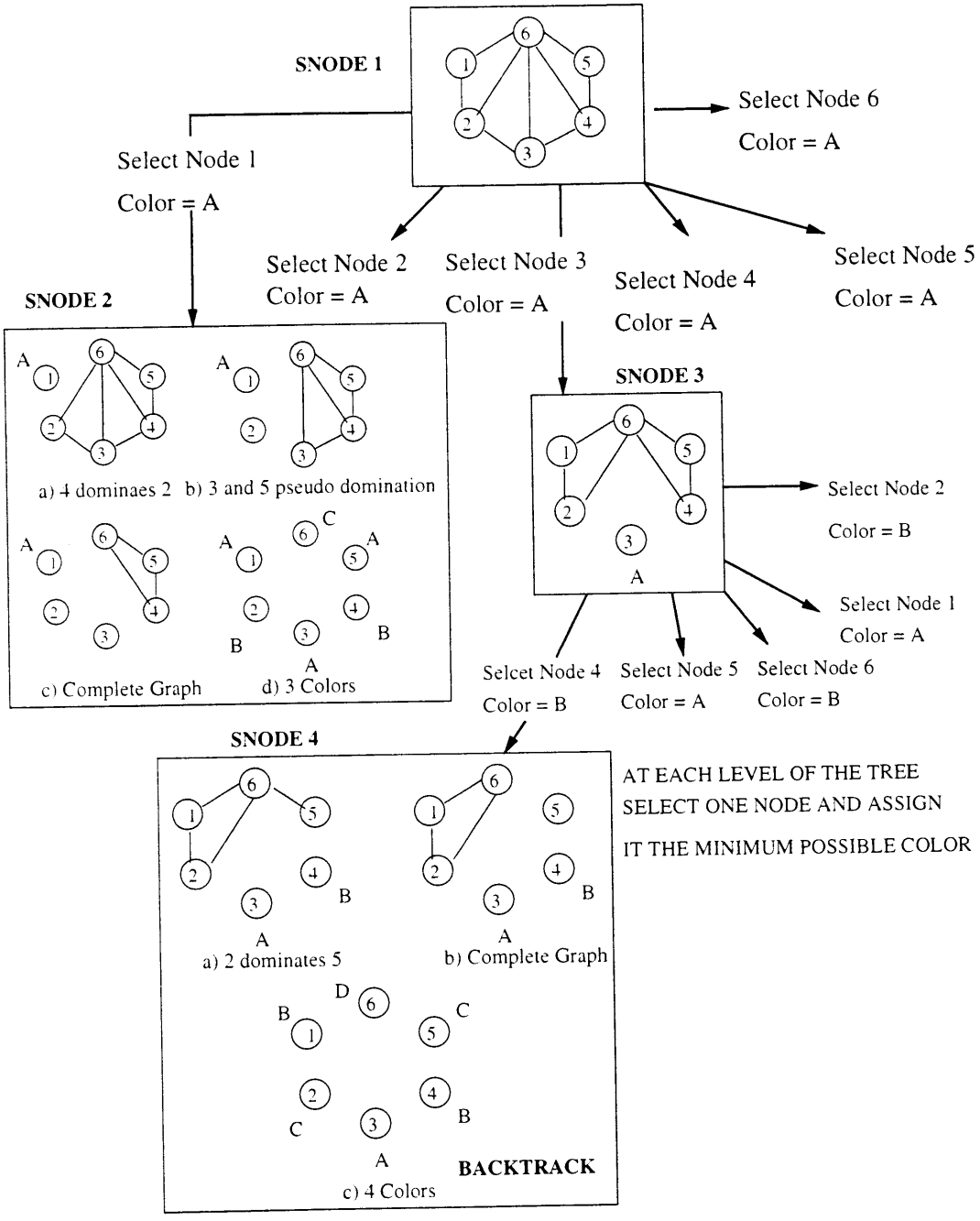


Figure 3.7: The tree Search showing how *DOM* can be made exact when it encounters a cyclic graph

Color C: {6}.
Backtrack to SNODE 1.

3. Now in SNODE 1, a new node in the graph which has not been selected before is selected. In any SNODE nodes are selected in a sequential order, until all the nodes in that SNODE have been selected. The next node selected in SNODE 1 will be Node 2, but this is not shown in the Figure 3.7. Instead, the next path of the tree, which is selecting Node 3 in SNODE 1 is shown.

4. On selecting Node 3 in SNODE 1 and giving it color A, a new graph is arrived at, shown in SNODE 3. This graph is still a cyclic graph. So select the first node in this graph, which is Node 4.
   Node 4 is assigned the minimum possible color which is color B. So in SNODE 4 the coloring led to four colors. The color assignment is:
   Color A: {3},
   Color B: {1,4},
   Color C: {2,5},
   Color D: {6}.
   Since this solution is worse the the previous solution, discard this solution.
   Backtrack to SNODE 3.

5. In SNODE 3 the next node choice is Node 5. So go along this new tree path, and repeat for all the nodes in SNODE 3. Repeat this process for all the nodes in all the SNODEs(for which *check_dominations* returned FALSE), at every level of the tree. If at any stage a better solution is found save the new solution and discard the old solution. For simplification, Figure 3.7 does not show the entire search space.

3.3.1 Algorithm for *EXDOM*

The Algorithm for *EXDOM* has been divided into two parts:

1. Function *check_dominations* which checks for dominations in the graph, it has two possible return values.

   (i) Returns "TRUE" if the graph was complete.

   (ii) Returns "FALSE" if the graph was cyclic, or disjoint.

2. Function *graph_coloring* is the top level function, which calls the function *check_dominations*.

Function *check_dominations* is the same as in *DOM* which was explained in Section 3.2.4.3. The following is the Algorithm for the function *graph_coloring*.

3.3.1.1 Algorithm for Function *graph_coloring* for *EXDOM*

NODES	is the set of nodes.
NODE_COUNT	is the count of the nodes in the graph.
DOM_COUNT	is the count of the dominated nodes.
CYC_NODES	is the number of nodes in the cycle.
CYC_NODES	is equal to *NODE_COUNT* - *DOM_COUNT*.
SN	is the selected node.
BEST_SOLN	is the number of colors assigned, initially set to be equal to *NODE_COUNT*
NEW_SOLN	is the new number of colors.

Step 1. **Initial Check to see if depth first search needed**:

Call function *check_dominations* to check for dominations. Every time function *check_dominations* finds a domination the dominated node is removed from the graph, and the count of dominated nodes removed (*DOM_COUNT*) is incremented. *check_dominations* returns TRUE if graph was complete, or FALSE if graph was not complete.

if (*check_dominations* == TRUE) /* If return value is TRUE */

store the solution and exit.

else

go to Step 2.

Step 2. **Creating the Initial State:**

Find $CYC\_NODES = NODE\_COUNT$ - DOM_COUNT

$SN = CYC\_NODES[0]$ Select one node from cyclic nodes.

Choose minimum possible color for SN.

Remove SN from the graph.

$CYC\_NODES = CYC\_NODES$ - 1 /*Decrement count of cyclic nodes*/

Go to Step 3.

Step 3. If ($CYC\_NODES \neq \emptyset$)

/* Call function check dominations */

if ($check\_dominations$ == FALSE)

go to Step 2.

else

calculate NEW_SOLN

if ($NEW\_SOLN < BEST\_SOLN$)

$BEST\_SOLN = NEW\_SOLN$ /* Save new best solution */

backtrack.

else

backtrack.

Theorem 3.2 *The search strategy shown for* EXDOM *in Section 3.3 will always be able to generate the exact solution.*

Proof 3.2 *This search strategy is a depth first search with one child, which means that for any node of the tree only one child is generated at a time, and each child of a parent node is deleted on returning to the parent of the node. At each level of the tree a different node in the cycle at that level of the tree is selected, and it is assigned the minimum possible color. Thus on reaching the leaf nodes of*

the tree all the possible solutions for the graph will be arrived at. The best of these solutions will be the chromatic number of the graph being colored. Thus this proves Theorem 3.2

By this search many best solutions for the graph may be arrived at but only the first best solution is saved. This method has not been programmed yet, hence there are no results for the method.

3.4 A New approach to an Exact Graph Coloring

Since Graph Coloring is an NP Complete problem, in general nearly all the possible solutions have to be evaluated, in order to find the best solution. The algorithm used here for the exact Graph Coloring is a greedy algorithm with backtracking and cut-off. This will be explained in more detail in this section. Here a new algorithm for coloring an incompatibility graph, with the minimum number of colors possible for this graph is presented. This algorithm is called *EXOC*, which stands for *Exact One Child.* In all latter sections the algorithm will be referred to by this name. *EXOC* uses a tree search in order to color the graph, and colors successively nodes with an actually available color of a smallest number, remembering for each node all the remaining possibilities of coloring (it is assumed that initially the set of colors has as many elements as the set of nodes).

Definition 3.6 *The chromatic number of a graph is defined as the number of colors in the exact solution.*

EXOC uses a "depth-first with one child" strategy. In this strategy at every stage only one branch of the tree is generated, and after finding a solution in a new branch, the solution that is better than a previous one, *EXOC* has a new, improved evaluation of the chromatic number. Thus, whenever a new solution is found, any color greater than the best solution is removed from the possible color choices of the other nodes. This best solution is used as the cut-off criterium. In any branch if the numbers of colors used is greater than or equal to the best solution, then *EXOC* does not proceed along that path any more and cuts-off. The difference between the "depth-first with one child" and a "depth-first search" is that in a "depth-first search" all the branches of the tree are created, all these branches are stored in memory, and each branch is followed until a leaf node of the tree is reached. But in the "depth-first with one child" whenever a new branch of the tree is created, only if its solution is better than the previous solution is the branch kept in memory and the old branch is deleted from memory. Also in

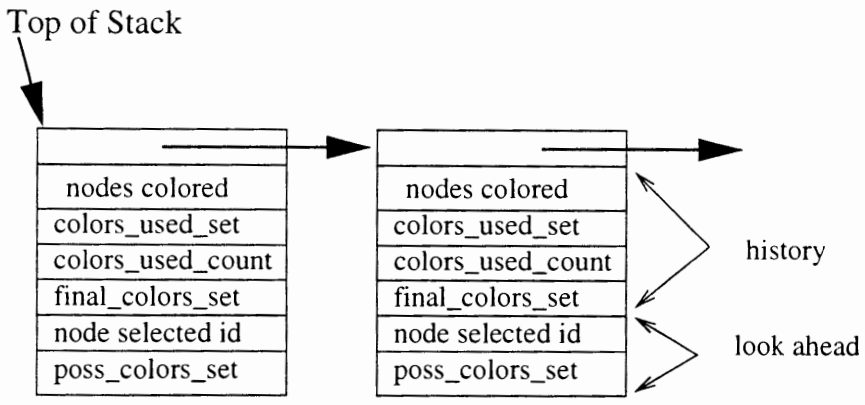


Figure 3.8: Data Structure for a Stack Node used in the Exact Graph Coloring Program

the "depth-first with one child" strategy, because there is a cut-off, *EXOC* does not go down to the leaf nodes of each branch.

3.4.1 The Exact Graph Coloring Program Implementation Details

EXOC was written in ANSI C on a SUN SPARC workstation. Two versions were programmed, one a stand-alone version and one which could be put into the Decomposition programs. Looking at the algorithm it could be seen that a "depth-first search with one child" strategy was needed, because it would provide the ability to cutoff and backtrack. The same would not have been possible with a breadth first search.

There were two possible approaches to implement the "depth-first search with one child" strategy, namely an approach using recursion, or an iterative approach using STACKS. A version using STACKS was chosen here. A STACK is a Last_in_First_out data structure, meaning the last information stored, is the first information which will be retrieved from the STACK.

One STACK is used in the program, the STACK is defined here as a singly linked list structure. Fig 3.8 shows the data structure used to implement the stack. The information held by each stack node can be divided into two parts. The **history** holds the information about what has happened before this node,

and the **Look Ahead** holds the information about the next node. The **history** holds four pieces of information: **nodes_colored** is the number of nodes colored before this stage, **colors_used_set** is the different colors used before this node, **colors_used_count** is the number of colors used until this node, and **final_colors_set** is the set of nodes and the colors assigned to the nodes until this node. The **Look ahead** holds two pieces of information: **node_selected** which is the node selected for the next node in the tree, and **poss_colors_set** which is the set of colors that the node selected can be colored with.

There are three main operations done on the STACK which are a PUSH, POP and PEEK. PUSH puts a new node on top of the STACK, and returns the new top of the stack. POP removes the topmost node, frees the memory for that node and returns the new top of the STACK. PEEK looks at the top of the stack, for some information about the topmost node.

In *EXOC* there are two possible ways to choose the node order, nodes can be chosen in a simple binary order and nodes can be chosen after sorting them in descending order according to the number of neighbors.

Theorem 3.3 *The number of colors generated for an incompatibility graph by* EXOC *will always be equal to the chromatic number of the graph.*

Proof 3.3 *For an incompatibility graph with* **n** *nodes, each node has at most* **nc** *possible colors that it can be colored where* **nc** $=$ **n**. *Hence if a depth first search is executed in which a different node is colored at different levels of the tree with all colors in* **nc** *which are possible for that node (possible color for a node means none of its neighbors can have the same color) then different solutions will be obtained at the leaves of the tree. The least cost of these solutions will be the chromatic number of this incompatibility graph. Many best colorings of the graph may be arrived at by this search. In the "depth-first with one child" search, for each node all the possible colors in* **nc** *for that node are not being checked but only a subset of colors in* **nc** *are being checked, but this subset of colors is initially equal to* **n**, *and when a solution* **new_sol** *is found, then only the number of colors in* **nc**

which is less than or equal to **new_sol** *is checked. Hence the "depth-first with one child" search will always arrive at one of the leaves of the tree which is the chromatic number of the graph, but will not find all possible minimum colorings for the graph. Hence this proves the Theorem 3.3, that the solution generated by* EXOC *is always the exact solution.*

Since *EXOC* cuts-off, whenever it finds in a branch of the tree that the number of colors used is equal to or greater than the best solution, the maximum memory of stack nodes used at any one time will be equal to the number of nodes in the incompatibility graph. The speed will depend on when the exact solution is found, in the worst case the exact solution will be found in the leaf node of the last path of the tree traversed, but this will rarely happen, if ever. In the best case the exact solution will be found in the leaf node of the first path of the tree traversed. In the program, to increase the efficiency, the step of cutting-off before calculating **poss_colors_set** for a node has been applied. From the candidates to color the selected node SN, one can select the colors which are different from colors used for nodes that have already been colored, if an edge exists between the SN and an already colored node. In the moment of finding solution in a node it is known that the minimal solution needs at most as many colors as in **colors_used_set**. Thus once a solution is found, from all **poss_colors_set** of all nodes any color which is $>$ **colors_used_count** can be removed, since we are not interested in all possible colorings, but only in the colorings with the accuracy to an isomorphism (i.e the minimum chromatic number colorings). Let us note that two types of nodes are dealt with here: nodes of the graph, and nodes of the search tree.

3.4.2 An Example showing how *EXOC* colors a graph

Figure 3.9 is an example which shows how *EXOC* colors a graph. In this example the incompatibility graph being colored is a non cyclic graph, but the algorithm for *EXOC* does not consider if the graph is cyclic or not, it uses the

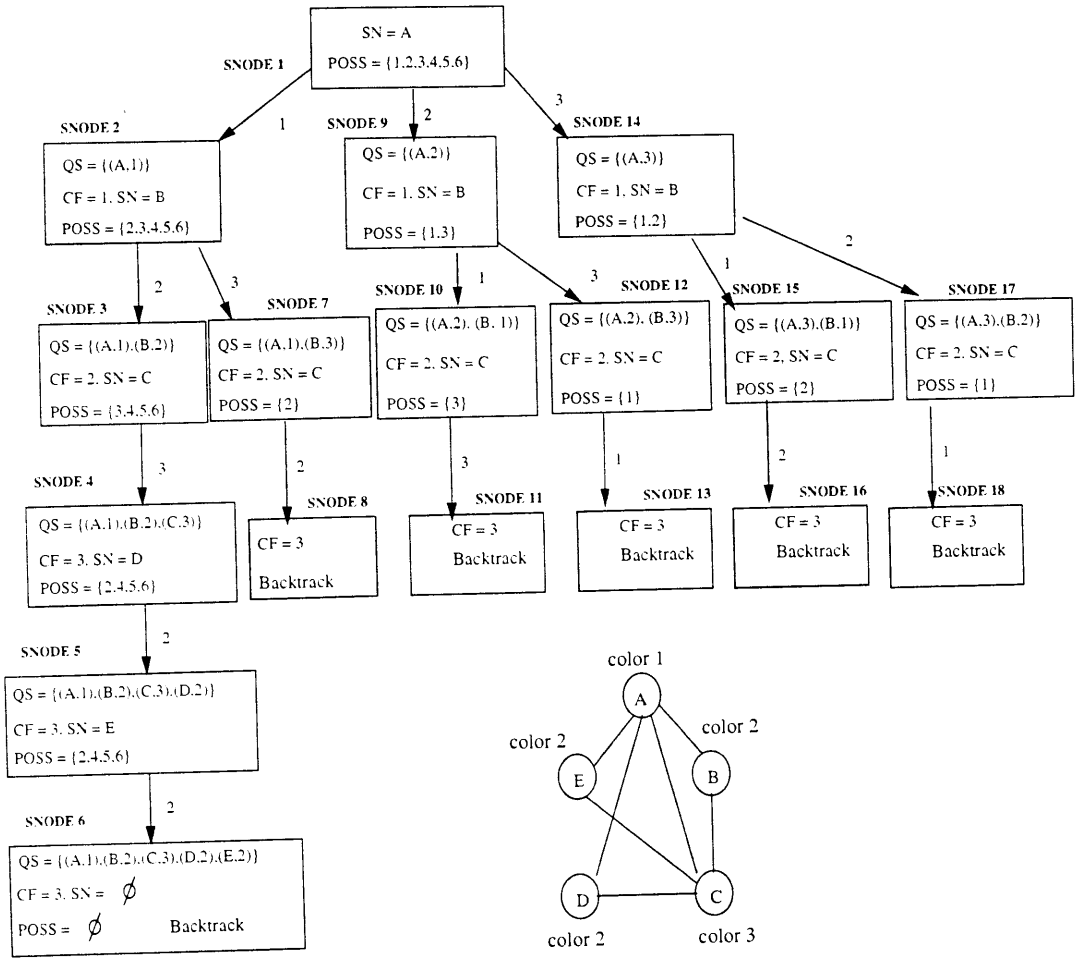


Figure 3.9: Tree Search for the Exact Graph Coloring Algorithm

same method to color a graph independent of whether the graph is cyclic or not. Since $EXOC$ deals with stack nodes and with nodes of the graph, in each step given below $SNODE$ refers to a node of the STACK and $GNODE$ refers to a node of the incompatibility graph.
The search process is executed as follows.

1. Select $GNODE$ "A"and find the possible colors for it and push the $SNODE$ labeled SNODE 1 in Figure 3.9 on to the stack.
   Selected color for $GNODE$ "A" is 1.

2. Now select a new $GNODE$ which is node "B". $GNODE$ "B" can be given the possible colors $2, 3, 4, 5, 6$. Store this information on $SNODE$ labeled SNODE 2 in Figure 3.9.
   Selected color is 2.

3. Now $GNODE$ "C" becomes the selected node and it can obtain one of colors {3,4,5,6}.
   Selected color is 3.

4. In this way, the first branch of the tree from Figure 3.9 is created.

   After finding the first solution:

   color 1 = {A},

   color 2 = {B,D,E},

   color 3 = {C}.

   we know that three colors are enough, and these colors are 1, 2 and 3.

5. All non-used colors are then removed from sets $POSS$ of all previous nodes. Now backtrack, and at each stage check the $POSS$ colors of the node. If there is a $POSS$ color then go along the new path, else ignore and continue backtracking.

6. On reaching $SNODE$ 2 there is a $POSS$ color = 3.
   Select this color for $GNODE$ B, and go to SNODE 7.

7. IN $SNODE$ 7, $GNODE$ "C" is the selected node, and it has only one possible color, which is color 2. Select color 2, for $GNODE$ "C" thus reaching $SNODE$ 8.

8. In $SNODE$ 8, CF = 3, which is the same as the best solution, which indicates that any solution along this path will at most result in a solution equal to the best solution, but not better. So cutoff and backtrack.

9. Ultimately on reaching $SNODE$ 1, select color 2 for $GNODE$ "A" and go along that path. On reaching $SNODE$ 11 the CF is 3, so do not proceed any more along this path, and cut off here and backtrack.

10. Figure 3.9 shows all the paths of the tree that are traversed. The $SNODEs$ are labeled in the order in which they are visited.

11. If at any SNODE a solution is obtained that is better than the one obtained at SNODE 6, store the new solution and discard the old solution. Continue till all the possible paths of the tree have been traversed. The solution saved is the minimum coloring of the graph.

3.4.3 Algorithm for *EXOC*

$NODES$	is the set of nodes.
$POSSIBLE\_COLORS\_SET$	is the set of colors possible for a node.
$COLORS\_USED\_SET$	is the set of colors used.
$COLORS\_USED\_COUNT$	is a count of the number of colors used.
$FINAL\_COLORS$	is a set containing ($NODE$, $COLOR$) for every node.
$COLORS\_USED\_MIN$	is the minimum number of colors.
$NODE\_COUNT$	is the number of nodes in the graph.

SN is the selected node.

Step 1. **Creating the Initial State:**

a) $NODE_1 := NODES[0]$, /* take first node from $NODES$ */.

b) Find $POSSIBLE\_COLORS$ for the node.

c) Remove $NODE_1$ from $NODES$.

d) $COLORS\_USED\_MIN = NODE\_COUNT$ /* Initial set up */

Step 2. **Depth First Search**

a) $SN :=$ first element from $NODES$.

b) Choose minimum $COLOR$ from $POSSIBLE\_COLORS$ for the node.

c) Remove SN from $NODES$

d) if ($COLOR \notin COLORS\_USED\_SET$) then
$COLORS\_USED\_COUNT$++.
Add new $COLOR$ to $COLORS\_USED\_SET$.
Add ($NODE$, $COLOR$) to $FINAL\_COLORS$.

e) else /* No Change in cost */
Add ($NODE$, $COLOR$) to $FINAL\_COLORS$.

f) if ($COLORS\_USED\_COUNT \leq COLORS\_USED\_MIN$) then

i) if ($REMAINING\_NODES$) == $\emptyset$)
/* Found better solution */
Store new best solution.

ii) else /*Still a possibility of better solution */
(continue).

g) else /* No chance for better solution along this path */
(cut-off, backtrack).

3.4.3.1 Pseudo code for *EXOC*

Explanation of Variables used in the Pseudo Code of *EXOC*

Variable Name	:*stack_top*
Description	:Data Structure of type shown in Figure 3.8, it is the pointer to the top of the STACK.
Variable Name	:*node*
Description	:Data Structure of type shown in Figure 3.8, it is a pointer to a single node in the STACK link list.
Variable Name	:*node_order*
Description	:Pointer to an array of integers, holds the order in which nodes are selected.
Variable Name	:*flag*
Description	:flag = NO_BEST_SOL means no best sol has been found :flag = BEST_SOL means best sol has been found.
Variable Name	:*best_sol*
Description	:Integer containing the number of colors in the best solution.
Variable Name	:*colors_used_count*
Description	:Integer count containing the number of colors used in a stack node of the tree.
Variable Name	:*final_colors*
Description	:Array of pointers of nodes to colors assigned to the

nodes.

Explanation of Functions called by *EXOC*

Function	:*sort_nodes.*
Synopsis	:sorts nodes according to the number of neighbors.
Returns	:array of type integer, containing the sorted nodes.
Function	:*find_possible_choices*
Synopsis	:Finds the possible colors for a node of the graph.
Returns	:pointer to array of type integer, containing the possible colors for the node.
Function	:*push_stack*
Synopsis	:Pushes a stack node on to the stack.
Returns	:Pointer to new top of the stack.
Function	:*possible_choices*
Synopsis	:Checks if there are any possible color choices for the graph node.
Returns	:TRUE if there is a possible color. :FALSE if there is no possible color.
Function	:*stack_peek*
Synopsis	:Copies history (shown in Figure 3.8) onto a new data structure of type stack node.
Returns	:Pointer to the new stack node.
Function	:*exchange_colors*
Synopsis	:Copies new colors assigned to nodes to *final_colors*

and removes the old color information.

Returns :New array of pointers of nodes and colors.

Function :*stack_pop*

Synopsis :Removes the top most node from the stack and frees the memory used for this node.

Returns :Pointer to the new top of the stack.

```
/* Initialize stack to NULL */
stack_top := NULL;
/* Decide a node order */
node_order := sort_nodes();
/*Select the first node */
node := node_order[0];
/* Find the possible colors that it can be colored */
node→possible_choices := find_possible_choices();
/* Push node onto stack */
stack_top := push_stack(node);
/* Now we are ready to do the depth first search */
while ( stack_top ≠ NULL ) {        /* Peek at top of stack */
      while ( possible_choices() ≠ 0 ) {
            /* Peek at top of stack and copy history on to new node */
            node := stack_peek();
            /* Find possible colors for the node selected */
            node→possible_colors := find_possible_colors();
            if ( flag == NO_BEST_SOL ) { /* We don't have a best sol yet */
                  if (nodes_colored == node_count) { /* We found a best sol */
                        /* Get the best solution */
                        best_sol := colors_used_count;
```

```
                final_colors = exchange_colors();
                flag := BEST_SOL;
              /* Set the color chosen to be 0 so it is not selected again */
                stack_top→possible_colors[color_chosen] := 0;
                stack_top := push_node(node);
        }
        else { /* All the nodes have not been checked yet */
                /* Set the color chosen to be 0 so it is not sel again */
                stack_top→possible_colors[color_chosen] = 0;
                stack_top := push_node(node);
        }
else { /* So we have a best sol already */
        if ( colors_used_count < best_sol ) {
                if (nodes_colored == node_count) {
                        /* So we have a better solution */
                        /* Get the new best solution */
                        best_sol := colors_used_count;
                        final_colors := exchange_colors();
                      /* Set the color chosen to be 0 so it is not sel again */
                        stack_top→possible_colors[color_chosen] = 0;
                        /* Get new stack top */
                        stack_top := push_node(node);
                }
                else { /* Still possibility of better sol exists */
                      /* Set the color chosen to be 0 so it is not sel again */
                        stack_top→possible_colors[color_chosen] = 0;
                        stack_top := push_node(node);
                }
         else { /* So there is no possibility of better sol */
  /* But we still have to remove the color tried from possibilities*/
```

```
                    stack_top→possible_colors[color_chosen] := 0;
                }
            }
        } /* End of while */
/* Free a node from top of stack and get new top */
stack_top := pop_node(stack_top);
}
```

3.4.4 Future Possible Improvements in *EXOC*

1. **Theorem 3.4** *If two nodes having an edge between them are chosen in the incompatibility graph and given different colors, then there is no need to branch for these two nodes and the solution will be exact.*

   Proof 3.4 *If in an incompatibility graph two nodes having an edge between them are chosen, then they have to be given different colors, and this is not going to change the number of colors in the graph because in all possible colorings of the graph these two nodes have to have different colors.*

   In the original Algorithm presented by Dr.Perkowski, the first stage started off with 2 nodes already being colored, and for the first node in the tree the selected node is the third node in the graph. But it was found that if these two nodes did not have an edge between them then if a bad color choice is made for these first two nodes then the algorithm will not generate the best solution. As no branching is done for the possible colors of these first two nodes the solution will not be exact if a bad color choice is made for the first two nodes. But as was proved above in Proof 3.4, if the two nodes chosen initially have an edge between them then they can be given different colors and it is not necessary to branch down the tree for any of these initial nodes, hence this is a possible future improvement in the *EXOC* program. In the original Algorithm presented by Dr.Perkowski, nodes were selected in

a sequential order, but in *EXOC*, nodes are sorted according to the number of neighbors, and then chosen one by one, instead of being chosen in a sequential order.

2. *EXOC* can be modified to have a smarter method of choosing the nodes. Possibly the node density or other local graph node-related parameters can be used to evaluate the order, or perhaps the cycle measures or other local graph cycle-related parameters can be used to evaluate the order. When *EXOC* is used in decomposition, depending on the type of decomposition that is being sought, the upper bound of the number of colors can be passed to *EXOC*. Therefore *EXOC* knows what the upper bound of the best solution is, before starting and *EXOC* will only generate a solution less expensive than the upper bound of the chromatic number if one exists. This would be a big speed up because now *EXOC* has the upper bound of the minimum solution even before the first pass. If a Curtis binary Decomposition is being sought, then the upper bound on the number of colors is always known. For example if there are 32 nodes in the graph, then it is known that the Bound Set had 5 variables. Now if there are 5 variables in the Bound Set then for a Curtis Decomposition to exist the number of output wires from the G block must be four or less than four. Now to get an output of four wires the number of colors in the graph of 32 nodes must be no more than 16 for a Curtis Decomposition to exist. Hence for this graph of 32 nodes, number 16 is the upper bound of the number of colors which can be passed to *EXOC* thus speeding up *EXOC*.

3. This algorithm can be modified to make it a Multi-Coloring. In Chapter 6 a new heuristic approach to a Multi-Coloring is presented. If *EXOC* was made into a Multi-Coloring, then it would be a Multi-Coloring with the least possible number of colors. Multi-Coloring will be explained in more detail in Chapter 6.

3.5 A New idea for an Exact Graph Coloring Program combining both *DOM* and *EXOC*

As was shown earlier in this Chapter, *DOM* generates the minimum number of colors whenever the resulting incompatibility graph after the check for dominations is a complete graph. Hence it was decided to combine *DOM* and *EXOC* together to have a new improved exact graph coloring algorithm. This new algorithm has been given the name *DOMEXOC*. The advantage of *DOMEXOC* over *EXOC* is that it will only call the depth first with one child search of *EXOC* if the check for dominations did not find a complete graph, thus it will be much faster than *EXOC*. Also if the check for dominations finds dominations in the incompatibility graph it will remove these dominated nodes from the graph, and hence a depth first search will now be done on a new reduced graph. In order to properly present the algorithm for *DOMEXOC* first an example is presented which shows how *DOMEXOC* colors a graph, and then the algorithm for *DOMEXOC* is presented.

3.5.1 Example showing how *DOMEXOC* colors a graph with cycles

1. Figure 3.10(a) shows the incompatibility graph of a function F. On checking for dominations in this graph it is found that Node 4 dominates node 6 and pseudo dominates Node 2. Hence Node 6 and Node 2 are removed from the graph. The function which checks for dominations now returns FALSE, cause it cannot find any more dominations in the graph, and the graph is not a complete graph.

2. The new resulting graph is shown in Figure 3.10(b). Now the "depth first with one child" strategy is applied to color this graph. The "depth first with one child" finds the exact minimum number of colors for the graph, resulting in the new graph shown in Figure 3.10(c).

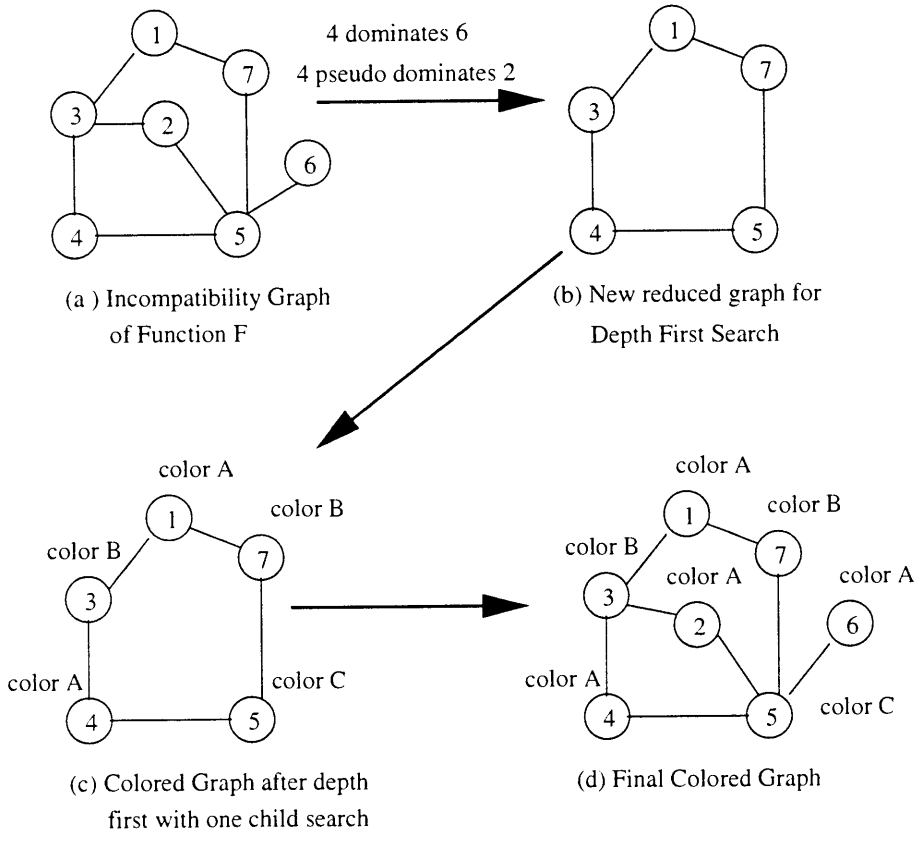


Figure 3.10: An Example showing how *DOMEXOC* colors a cyclic graph

3. Now the dominated nodes 2 and 6 are colored with the same color that was assigned to the node 4 which is the node that dominates them. The final color assignment is:
   Color A : {1, 2, 4, 6},
   Color B : {3, 5},
   Color C : {5}.

4. As can be seen from the above example the "depth first with one child" has to color a graph with five nodes instead of the original graph with seven nodes.

3.5.2 Algorithm for *DOMEXOC*

The Algorithm for *DOMEXOC* uses the function *check_dominations* used by *DOM*, and the function used by *EXOC* to do the depth first search. Since these algorithms were explained in detail in earlier sections, they will not be repeated here. The Algorithm explained here for *DOMEXOC* is the top level Algorithm.

NODES : is the number of nodes in the incompatibility graph.
CYC_NODES : is the nodes remaining after function *check_dominations* has removed some dominated nodes from the incompatibility graph.

Step 1. Call function *check_dominations*.
   if (*check_dominations* == TRUE) /* Is Graph Complete */
     go to Step 4
   else
     go to Step 2.

Step 2. Since *check_dominations* has already removed the dominated nodes, call the "depth first with one child" search with the new graph. The number of nodes in this new graph is equal to *CYC_NODES*. The "depth first with

one child" search will return the colored graph.

Go to Step 3.

Step 3. if (*CYC_NODES == NODES*)

```
    /* This means there were no dominations found
     Hence the graph is already colored. */
     exit(0); /* Exit from program */
else /* This means some dominations were found in the graph */
     /* Color the dominated nodes the same color as the nodes which
        dominated them */
     color_graph();
     exit(0); /* Exit from program */
```

Step 4. If we arrived here it means that the function *check_dominations* found a complete graph. So just color the graph, giving the dominated nodes the same color as the nodes which dominate them.

```
color_graph()
exit(0); /* Exit from program */
```

Since *DOMEXOC* combines the features of both *DOM* and *EXOC* it should be much faster than *EXOC* and will only use the "depth first with one child" when *check_dominations* returns FALSE. *DOMEXOC* has not been programmed yet and hence there are no results for this method. This idea has been presented here as it was felt that this could be something which could be programmed by someone in the Functional Decomposition group in the future by using both the existing programs *DOM* and *EXOC*.

3.6 Summary and Conclusions of Chapter 3

After implementing these two graph coloring programs, *DOM* and *EXOC* which were presented in this Chapter, we needed to compare these two programs. In the next Chapter *DOM* and *EXOC* are compared on randomly created

graphs. In Chapter 5 a comparison is made between the *DOM*, the greedy Clique Partitioning, and the *EXOC* Algorithms in Decomposition applications.

CHAPTER 4

A COMPARISON OF THE *DOM* AND *EXOC* GRAPH COLORING PROGRAMS ON RANDOM GRAPHS

In this Chapter an evaluation is done to see how the heuristic Dominance Graph Coloring(*DOM*) and the Exact Graph Coloring(*EXOC*) perform on randomly generated graphs. In order to do the evaluation a random graph generator program was written. This random number generator is called *RANG* [1]. *RANG* accepts as its arguments the number of nodes and the percent of edges, and generates a graph. The system time is used as the seed for *RANG*, thus ensuring that the graphs generated are totally random. In this Chapter first the *DOM* program was run on randomly generated graphs, and an evaluation of how well *DOM* performs in terms of time of execution was done. Then both *DOM* and *EXOC* were run on the same randomly generated graphs and their results were compared. These evaluations were done to see how *DOM* and *EXOC* perform with respect to increasing numbers of nodes with varying percent of edges, and to see how close to exact are the solutions generated by *DOM* on the randomly generated graphs.

4.1 Results of running *DOM* on Randomly generated Graphs

The results of running *DOM* on random graphs as a stand alone version are shown in Table 4.1. In the table "T" represents the user time and "C" represents the number of colors. These graphs have been generated with nodes from 100 to 900 with 10% to 90% of edges. These graphs were generated by the random graph generator. It was found that due to the random nature of the graphs,

[1] *RANG* can be obtained from the directory /stash/polo at Portland State University

Nodes vs Edges										
Edge%		*Nodes*								
		100	200	300	400	500	600	700	800	900
10%	T(s)	0.17	0.61	1.1	0.98	1.45	3.69	2.88	5.2	5.77
	C	6	10	13	16	18	21	23	26	28
20%	T(s)	0.2	0.41	0.71	1.25	1.98	3.99	3.96	7.31	8.97
	C	11	16	21	26	31	36	40	44	49
30%	T(s)	0.2	0.53	1.04	1.55	2.63	4.51	5.25	9.52	12.78
	C	14	22	30	36	44	50	59	62	69
40%	T(s)	0.23	0.52	1.14	1.78	3.03	4.51	6.6	11.9	16.49
	C	16	28	37	47	57	64	74	80	91
50%	T(s)	0.23	0.64	1.27	2.07	3.76	5.51	7.81	15.14	20.93
	C	21	34	48	60	73	83	94	104	115
60%	T(s)	0.30	0.82	1.32	2.40	4.76	6.51	9.8	18.41	25.38
	C	23	42	57	72	86	105	115	130	142
70%	T(s)	0.28	0.75	1.53	2.86	5.04	7.67	11.35	21.25	30.92
	C	28	49	69	92	110	122	140	159	176
80%	T(s)	0.28	1.03	1.83	3.26	5.39	8.32	12.59	26.35	37.95
	C	36	61	88	113	135	156	178	201	223
90%	T(s)	0.31	0.93	2	3.97	6.21	9.95	14.93	31.84	45.49
	C	45	78	114	142	176	203	231	259	290

Table 4.1: Nodes vs Edge Percent for *DOM* on randomly generated graphs

dominations are rarely found, which is the basis of *DOM*. Still *DOM* is fast, and on small graphs the results can be verified, but on large graphs the results cannot be verified.

To evaluate the efficiency of *DOM* a graph of Nodes vs time has been plotted which is shown in Figure 4.1 for different percentage of edges. Nodes are shown along the X axis, and time in seconds is shown along the Y axis. By statistical

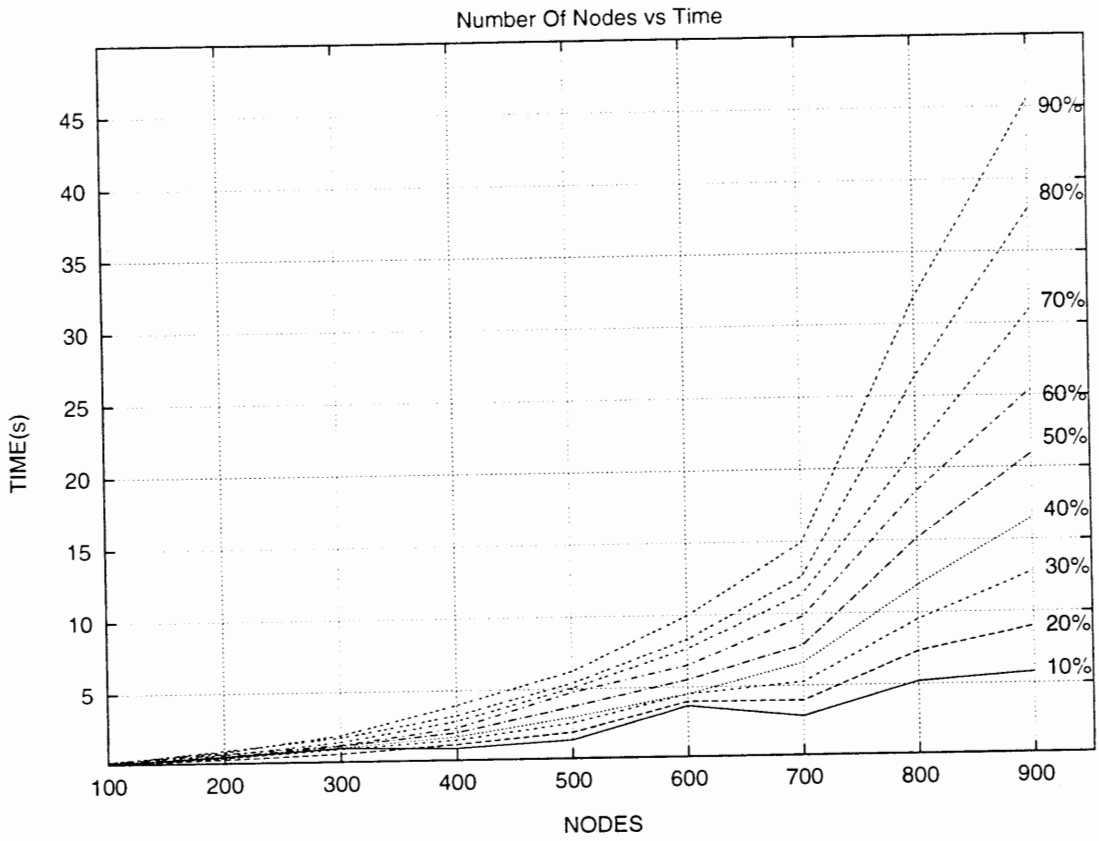


Figure 4.1: Plot of Nodes vs Time for *DOM* on graphs with increasing number of nodes with different percent of Edges

analysis it was found that $T \propto N^{4.3}$.

4.2 Comparison of *DOM* and *EXOC* on Randomly generated Graphs

In order to evaluate *DOM* in terms of the quality of solutions generated by it, a comparison was made by running both *DOM* and *EXOC* on the same graphs. The following Tables show the results of the testing.

4.2.1 Notations used in the Tables

E.P : Edge Percent

P : Program

E1 : *EXOC* with sorted order of nodes

E2 : *EXOC* with nodes in sequential order

D : *DOM*

T : Time in Seconds

C : Number of Colors generated

N.B : Number of backtracks required.

1 means best solution was found in the first pass,

2 means best solution was found in the second pass

(number) Number of colors *DOM* was away from the best solution

Edge percent from 10%-45% in steps of 5%													
EP	P	Nodes											
		10			20			30			40		
		T(s)	C	T.B	T(s)	C	T.B	T(s)	C	T.B	T(s)	C	T.B
10%	E1	0	2	1	0	3	1	0	3	1	0	4	1
	E2	0	2	1	0	3	1	0	3	2	0.1	4	1
	D	0	2		0	2		0	3		0	4	
15%	E1	0	3	1	0	2	2	0	3	1	0	5	1
	E2	0	3	1	0	2	2	0	3	1	0	5	1
	D	0	3		0	**3(1)**		0	3		0	5	
20%	E1	0	3	1	0	4	1	0	4	1	0	5	1
	E2	0	3	1	0.3	4	2	2.9	4	1	1.5	5	2
	D	0	3		0	4		0	4		0	**6(1)**	
25%	E1	0	3	1	0	4	1	0	5	2	0.6	6	1
	E2	0	3	1	0.4	4	2	10	5	1	180	6	1
	D	0	3		0	4		0	**6(1)**		0	**7(1)**	
30%	E1	0	3	1	0	4	1	0.7	5	1	10.5	6	2
	E2	81	3	1	2	4	2	40.7	5	3	15.9	6	2
	D	0	3		0	**5(1)**		0	5		0	**7(2)**	
35%	E1	0.1	3	1	0	4	2	3.5	5	1	39	6	1
	E2	0.2	3	1	205	4	2	79	5	3	249	6	3
	D	0	3		0	**5(1)**		0	**6(1)**		0	**8(2)**	

Table 4.2: Comparison of *EXOC* and *DOM* on randomly generated graphs with edge percent from 10 to 35

In Table 4.2 *EXOC* was run both with nodes in sorted order and with nodes in sequential order. In this Table, nodes were varied from 10 to 40 in steps of 10, and edge percent was varied from 10% to 35% in steps of 5%. The following is a summary of the results obtained in Table 4.2.

1. Total Number of Program Runs = 24.

2. The version of *EXOC* which selected nodes after sorting them according to the number of edges(E1), found the solution in the first pass in 20 cases, and in the second pass in 4 cases. The version of *EXOC* which selected nodes sequentially, found the solution in the first pass in 13 cases, in the second pass in 8 cases, and in the third pass in 3 cases.

3. *DOM* found the best solution in 15 cases, was one color away from the best solution in 7 cases and was two colors away from the best solution in 2 cases. The cases when *DOM* did not find the best solution are shown in bold in Table 4.2.

Edge percent from 40%-90% in steps of 10%										
EP	P	Nodes								
		10			20			30		
		T(s)	C	T.B	T(s)	C	T.B	T(s)	C	T.B
40%	E1	0	4	1	0.1	5	1	0.8	6	2
	D	0	4		0	5		0	**7(1)**	
50%	E1	0	4	1	1.5	6	1	1200	**8**	2
	D	0	4		0	6		0	**9(1)**	
60%	E1	0	5	1	6.8	7	1	12000	9	2
	D	0	5		0	7		0	**11(2)**	
70%	E1	0	5	1	16.8	**8**	2	18000	10	2
	D	0	5		0	**9(1)**		0	**12(2)**	
80%	E1	0	5	1	-	-		-	-	
	D	0	5		0	10		0	16	
90%	E1	0	7	1	3600	11	1	-	-	
	D	0	7		0	11		0	16	

Table 4.3: Comparison of *EXOC* and *DOM* on randomly generated graphs with edge percent from 40 to 90

In Table 4.3 *DOM* and the version of *EXOC* which selected nodes after sorting according to the number of edges(E1) were run, on graphs with nodes from 10 to 30 in steps of 10 and edge percent from 40% to 90% in steps of 10%. The following is a summary of the results of Table 4.3.

1. Total Number of Program Runs = 18.

2. The version of *EXOC* which selected nodes after sorting according to the number of edges, found the solution in the first pass in 10 cases, in the second pass in 5 cases, and *EXOC* was terminated in 3 cases, as *EXOC* took too long(time greater than 18000 sec).

3. *DOM* found the best solution in 10 cases, was one color away from the best solution in 3 cases, and was two colors away from the best solution in 2 cases.

4.3 Summary and Conclusions of Chapter 4

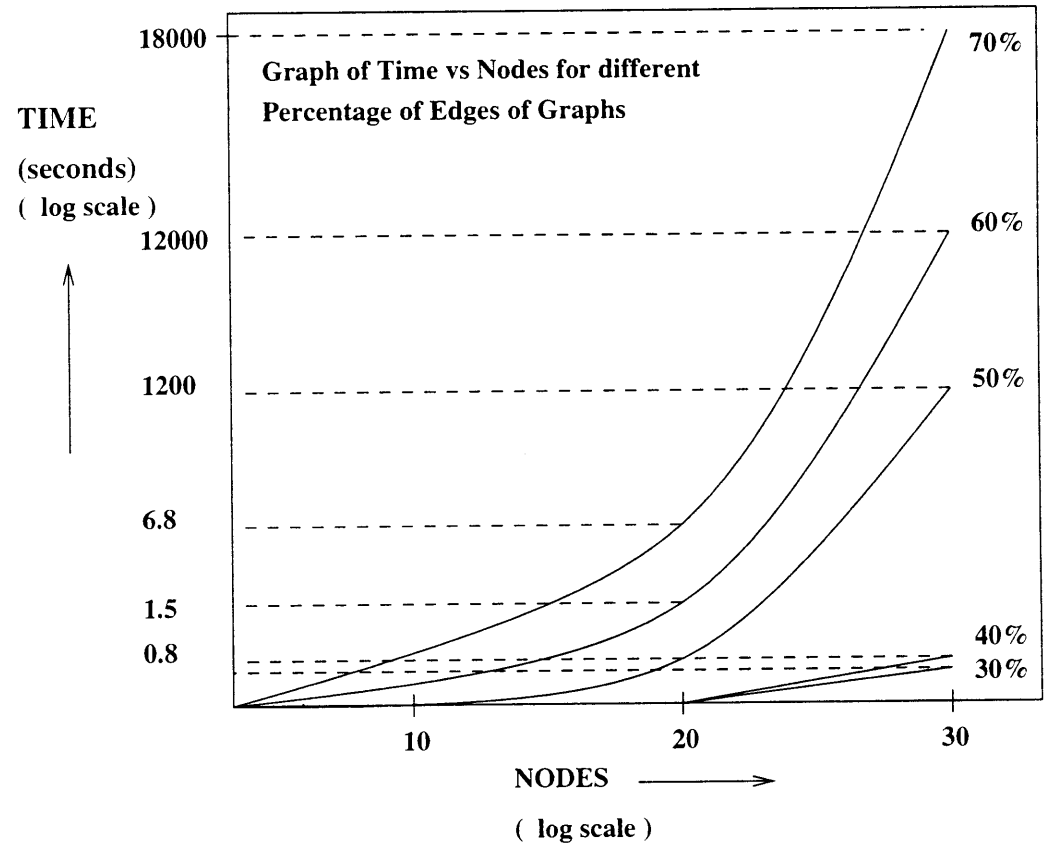


Figure 4.2: Graph of Nodes vs Time showing how *EXOC* performs on randomly generated graphs with edge percent from 30% to 70%

In order to evaluate the results obtained in Table 4.2 and Table 4.3, a graph of nodes vs time for varying percent of edges was plotted for *EXOC*. Figure 4.2 shows the plot. As can be seen from the plot as the number of nodes increases, for graphs having percentage of edges up-to 40% the time taken by *EXOC* is a maximum of 0.8 sec which is very fast. But once the percent of edges increases

Number of Program Runs = 39					
Times Sol Exact		Times Sol 1 color away		Times Sol 2 colors away	
Number	Percent	Number	Percent	Number	Percent
25	64%	10	25.6%	4	10%

Figure 4.3: Summary of Results showing how close to exact are the solutions generated by *DOM* on randomly generated graphs

to 50% the time *EXOC* takes to color a graph with 30 nodes and 50% of edges is 1200 seconds, and for a graph with 30 nodes and an edge percent of 60, the time increases by a 1000 times and becomes 12000 secs.

Observation 4.1 *When the chromatic number of a graph is high* EXOC *is going to be slow.*

To support this observation note that when there is a high percent of edges in a graph, the number of colors in the best solution is going to be high that is the chromatic number is going to be high. Hence each node is going to have a high number of possible colors, with the number of possible colors being highest for the node which is colored first. Now if there is a high number of possible colors, and if the best solution itself is high each possible color which is smaller or equal to the best solution has to be tried, and this is can be a very large search space. These results indicate that for graphs with 30 or more nodes and more than 50% of edges, it is not so important to have a good node order, or even to have the chromatic number passed into *EXOC*, but only if the graph has a low chromatic number will the speed of *EXOC* be faster 4.1. This suggests that *EXOC* should only be used for applications in which the size of the graphs are less than 40 nodes, and in the graphs having more than 30 nodes, the percentage of edges should be less than 50%.

Table 4.3 shows how close to exact are the solutions generated by *DOM*. As can be seen *DOM* generates the exact solution in 64% of cases, is one color away from

the best solution in 25.6% of cases, and in 10% of cases is two colors away from the best solution. Thus this suggests that, if the application does not require an exact solution at all times, and speed is the primary criterium, then *DOM* will be better than *EXOC*.

As the next step, we are going to test *DOM* and *EXOC* in Functional Decomposition. Since in Functional Decomposition, speed is of great importance to us, we know that if the graphs generated during decomposition fall in the category where they have a high percent of edges, then, even if *EXOC* generates better results than the heuristic methods, it will not be a practical method, since it will be too slow. This is presented in the next Chapter. In the next Chapter we also want to answer the following questions:

1. Does generating exact results for the graphs at every step of the Decomposition, improve the quality of the decomposition achieved?

2. Are the results generated by *DOM* for the graphs generated during the steps in Functional Decomposition for real life functions going to be better than the results generated for random graphs?

CHAPTER 5

AN EVALUATION OF THE PROBLEM OF COLUMN MULTIPLICITY IN DECOMPOSITION OF FUNCTIONS

In this Chapter we will evaluate the importance of an Exact Graph Coloring in Curtis Decompositions. Our aim is to investigate if an Exact Graph Coloring is required in Functional Decomposition and if it leads to better results on the graphs that are created from practical function benchmarks. We used the Decomposer Multi-Valued General Universal Decomposer*MVGUD* written at Portland State University for the testing purpose. We instantiated three algorithms into *MVGUD*, a Greedy Clique Partitioning, the Dominance Graph Coloring (*DOM*) and the Exact Graph Coloring(*EXOC*). The decomposer was run with different numbers of variables in the Bound Set on two kinds of benchmarks: *MCNC* benchmarks for circuits, and Machine Learning Benchmarks(from the Wright Labs Database) for data from Machine Learning, Pattern Recognition and Knowledge Discovery in Data Bases.

In the previous Chapter a comparison of *DOM* and *EXOC* was done on randomly generated graphs, for varying number of nodes and varying percentage of edges. Thus, in the previous Chapter, conclusions were reached about how well *DOM* and *EXOC* will perform on the different kinds of graphs. So here too tests were done to characterize the kind of graphs that are generated in decomposition with regard to the number of nodes in the graph and the percentage of edges in the graph in order to see if the same conclusions hold for the graphs generated during Functional Decomposition. In this Chapter first some notions and definitions are presented, then a brief introduction to *MVGUD* is presented, in which the strategy used by *MVGUD* to perform decompositions is explained and then the

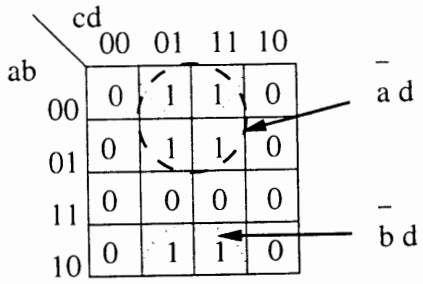


Figure 5.1: The Karnaugh map of a function showing a Vacuous Variable

evaluation of the Column Multiplicity problem in Functional Decomposition is presented.

5.1 Some Notions and Definitions

5.1.1 A Vacuous Variable

Definition 5.1 *A vacuous variable in a Function F is an input variable which is not needed to realize Function F. Removing a vacuous variable will not change the function.*

Figure 5.1 shows a Karnaugh map of a function having four binary inputs and one binary output. On solving the Karnaugh map it can be seen that input variable "c" does not appear in the solution, hence input variable "c" is not needed to realize the function, and it can be removed.

5.1.2 Clique Partitioning and Clique Covering

Since *MVGUD* has a greedy Clique Partitioning incorporated in it, here a Clique Partitioning, and the difference between a Clique Partitioning and a Clique Covering is presented.

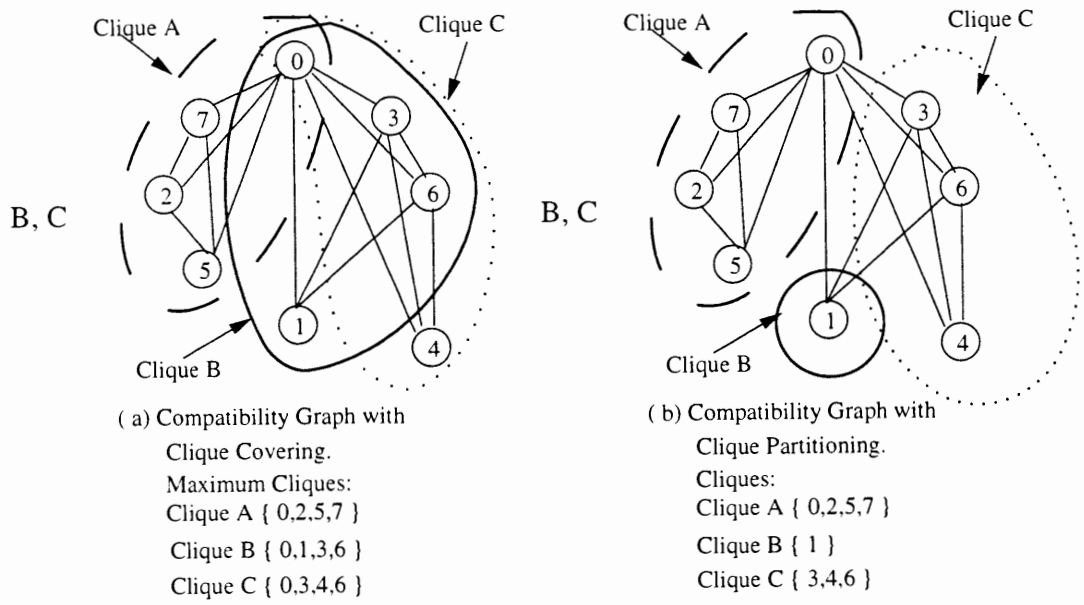


Figure 5.2: The Difference between Clique Covering and Clique Partitioning

5.1.2.1 Clique Partitioning of a Compatibility Graph

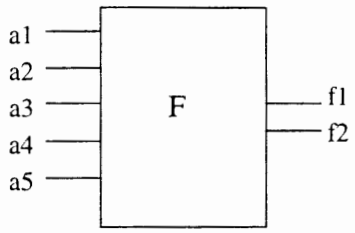
Definition 5.2 *Clique Partitioning of a Graph $G = (V, E)$ is a partitioning of the vertex sets of G into independent sets (color classes). These independent sets are called* **cliques**.

Clique Partitioning is done on an Compatibility Graph, in Clique Partitioning there is no overlap between the Cliques. Figure 5.2(b) shows a Clique Partitioning of a graph into three cliques, as can be seen there is no overlap between the cliques.

5.1.3 Clique Covering of a Compatibility Graph

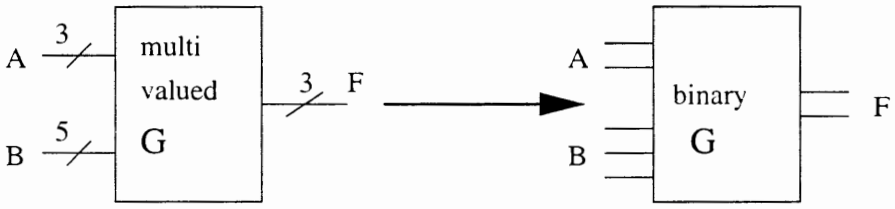
Definition 5.3 *Clique Covering of a Graph $G = (V, E)$ is to find a set of cliques of minimum cardinality whose union covers all the vertices of the graph.*

Figure 5.2(a) shows a Clique Covering of a graph. As can be seen here, there are still three cliques, but now there is an overlap between the cliques.



$DFC = 2^5 * 2 = 64$

(a) DFC Calculation in a Binary Function



$DFC = 2^5 * 2 = 64$

(b) DFC Calculation in a Multi-Valued Function

Figure 5.3: Calculation of DFC in Binary and Multi-Valued Functions

5.1.4 Decomposed Function Cardinality

Definition 5.4 *Decomposed Function Cardinality(DFC) for a binary function is a measure of a function complexity and is defined as* $2^N * M$, *where* N *is the number of inputs and* M *is the number of outputs of the binary function.*

DFC is defined as the cost of the minimum Decomposition. DFC is a good evaluation of the Decomposition quality for Machine Learning Applications but it does not take into account the complexity of the decomposed blocks. Figure 5.3(a) shows how DFC is calculated for a Binary Function.

For multi-valued functions, DFC has not been defined as a standard yet. In order to calculate the *DFC* of the resulting function, *MVGUD* translates the multi valued inputs and outputs of each block in the final Decomposition into their binary equivalent ($\log_2 multi\_value$) and then calculates the equivalent

binary DFC. In Figure 5.3(b) input variable A which is 3-valued, is converted to 2 bits, and input B which is 5-valued is converted to 3 bits. The output is converted to 2 bits. Then the formula used to calculate the binary DFC is used resulting in a DFC of 64.

5.2 An Introduction to MVGUD

MVGUD [26] is a multi-valued decomposition program written by Stanislaw Grygiel at Portland State University. The following algorithm explains the strategy used by *MVGUD* to perform multi-valued decompositions. This algorithm explains one step of the decomposition process.

5.2.1 Strategy used by *MVGUD* to perform Decompositions

Step 1 For an input function F: Generate a set of Disjoint Partitions:

$$PART\_SET = \{(B_1, A_1), (B_2, A_2), (B_i, A_i), \cdots (B_n, A_n)\}$$

where $n = 8 \cdots 16$, B_i is the Bound set and A_i is the free set of Partition i of the Function F. Size of the Bound Set is an input parameter to the program *MVGUD*. These partitions are generated using Wan Wei's [6] algorithms for Variable Partitioning.

Step 2 Set initial column multiplicity (μ_{min}) to a high number.

Step 3 Choose a partition (B_i, A_i) from $PART\_SET$.
Call the clique partitioning/graph coloring program to calculate the column multiplicity (μ_{B_i,A_i}) of chosen Partition $\{(B_i, A_i)\}$.

Step 4 If $\mu_{B_i,A_i} < \mu_{min}$ then set
$\mu_{min} = \mu_{B_i,A_i}$, go to Step 5.
If $\mu_{B_i,A_i} = 1$, break out and return 1 because this means that all the Variables in the Bound Set are vacuous.

Step 5 If all Partitions in $PART\_SET$ have not been tried, go to Step 3 else go to Step 6.

Step 6 Now check if the resulting μ_{min} is an acceptable Curtis Decomposition. In order to do this find the highest multi-value in the inputs of the Bound set $B_i = m$, and let the number of variables in the Bound set $B_i = n$. Here B_i is the Bound set belonging to the $PART\_SET = \{(B_i, A_i)\}$ which resulted in the best μ_{min}. Now if $n > \log_m(\mu_{min})$ then it is an acceptable Curtis decomposition. This decision made by *MVGUD* to decide if a μ is an acceptable Curtis decomposition, can be better understood by considering a Binary decomposition. If a function F has n binary inputs, then for a Curtis Decomposition to exist $n > \log_2 \mu$. Hence the same reasoning is applied to a multi-valued function, only now the base of the log is the highest multi-valued input of the Function. If an acceptable decomposition was found then return with the decomposition found. If it is not an acceptable Curtis Decomposition, then increase size of the bound set and go back to Step 1.

Step 7 If the size of the bound set is one less than number of inputs and still no decomposition is found, go back to Step 1, only now generate-non disjoint partitions.

Since *MVGUD* is a multi-valued decomposer, it has no encoding stage. Essentially *MVGUD* looks for the Curtis Binary Decomposition criteria which was explained in Chapter 2 in evaluating if a decomposition is acceptable, but then assigns the output of the "G" function one value which is equal to the μ_{min} found. This approach results in one multi-valued output from each "G" block, which can be called a multi-valued Ashenhurst Decomposition. Whether the method used by *MVGUD* to calculate DFC (explained in Section 5.1.4), is a good evaluation of the cost of the decomposed multi-valued blocks is not known, but since the DFC is used for a comparison between different methods of calculating the Column Multiplicity in Decomposition, within the same decomposer, the method of

calculation of the DFC does not matter for the purpose of evaluating algorithms for calculating column multiplicity. What matters is that the same method is used for all the algorithms that are compared.

5.3 A Comparison of the different Strategies of finding the Column Multiplicity in Functional Decomposition

How the different algorithms for calculating the Column Multiplicity in Functional Decomposition compare on functions from Circuit, and Machine Learning Benchmarks will be presented in this section. Here a comparison is done between *DOM*, *EXOC* and the heuristic Clique Partitioning Program which we will call *CLIP*. The goal of this testing is to see if an Exact Graph Coloring is necessary to calculate the Column Multiplicity in Functional Decomposition, and if the DFC can be improved in case that *MVGUD* is run with *EXOC*, in comparison to when it is run with *DOM* or with *CLIP*. The results of the testing have been divided into two parts, first the testing is done on the Circuit benchmarks, and then the testing is done on the Machine Learning Benchmarks. *MVGUD* was tested with two, four, and five variables in the Bound set.

5.3.1 Notations Used in the Tables

The following is an explanation of the Notations used in the Tables in this Chapter.

1. Benchmark : Name of the Benchmark function.
2. in : Number of inputs of the Benchmark.
3. out : Number of outputs of the Benchmark.
4. cubes : Number of cubes in the Benchmark.
5. DFC : Decomposed function cardinality of the decomposed function.

6. Algorithm : Name of Algorithm used in *MVGUD*.

7. Nu of Blocks : Number of multi-valued blocks in the decomposed function.

8. NP : *Number of passes*, or number of times the function to calculate the column multiplicity was called.

9. TC : *Total Colors*, iterative sum of colors generated for each pass.

10. AC : *Average Colors* = TC/NP.

11. T(s) : User time in seconds.

5.3.2 A Comparison of the different Strategies of finding the Column Multiplicity on MCNC Benchmarks

5.3.2.1 A Comparison when *MVGUD* is run with 2 variables in the Bound Set

Table 5.1 shows a comparison made by running *MVGUD* with 2 variables in the Bound Set for *EXOC*, *DOM* and *CLIP*. Comparisons were made with respect to the DFC, the number of two-input gates in the final decomposed function, and the time taken by *MVGUD* to decompose the function. In Table 5.1 *2 i/p g* stands for the equivalent number of two input gates. This is calculated by *MVGUD*.

As can be seen from Table 5.1, *DOM* and *CLIP* both perform well in different cases. *DOM* provides a smaller DFC in 6 cases and *CLIP* provides a smaller DFC in 5 cases, while *EXOC* provides a tie with the best solution in 6 of the cases. Hence *EXOC* does not provide any real improvement, on the other hand it is much slower than *CLIP* and *DOM*, which is to be expected. The reason for this kind of results is that since in this experiment there are 2 variables in the Bound Set, in most cases the size of the Incompatibility Graph at different levels of the decomposition is 4, which is a small graph, and it is known that for small graphs both *CLIP* and *DOM* usually generate the best solution as well. Hence

Comparison of *EXOC*, *DOM* and *CLIP* on MCNC Benchmarks											
Benchmark	in	out	cubes	Algorithm	DFC	nu of blocks	2i/p g	T(s)	NP	TC	AC
5xp1	7	10	143	*EXOC*	274	36	69	76	168	613	3.6
				DOM	274	36	69	10.5	168	611	3.63
				CLIP	314	34	79	10.6	163	589	3.6
9sym1	9	1	158	*EXOC*	112	4	28	103	38	146	3.8
				DOM	60	5	15	29.7	38	146	3.8
				CLIP	112	4	28	31.2	34	136	4
b12	15	9	172	*EXOC*	302	52	76	84	353	940	2.66
				DOM	302	52	76	12.2	353	940	2.66
				CLIP	302	52	76	10.9	351	924	2.64
con1	7	2	18	*EXOC*	90	11	23	8.3	58	230	3.9
				DOM	94	10	24	8.0	58	230	3.9
				CLIP	90	11	23	7.9	55	204	3.71
bw	5	28	97	*EXOC*	544	94	136	54	431	1446	3.35
				DOM	568	95	142	18.2	430	1443	3.36
				CLIP	532	95	133	18.2	426	1422	3.34
ex5p	8	63	214	*EXOC*	2058	372	505	889	1623	4897	3
				DOM	2058	373	515	160	1615	4853	3
				CLIP	2018	372	505	183.3	1611	4765	2.96
misex1	8	7	40	*EXOC*	274	34	69	41.1	170	593	3.4
				DOM	298	34	75	8.8	168	581	3.46
				CLIP	274	34	69	8.5	169	574	3.4
rd53	5	3	63	*EXOC*	72	10	18	7.9	30	91	3.3
				DOM	72	10	18	1.5	30	91	3.3
				CLIP	72	10	18	1.8	30	91	3.3
rd73	7	3	274	*EXOC*	148	15	37	64	52	161	3
				DOM	72	10	18	11.0	52	161	3
				CLIP	148	15	37	12.1	54	176	3.26
rd84	8	4	515	*EXOC*	248	24	62	213	76	234	3
				DOM	208	25	52	34.5	76	234	3
				CLIP	248	24	62	30.6	80	259	3.24
sao2	10	4	133	*EXOC*	436	31	109	2160	205	1051	5.1
				DOM	540	29	135	49	205	1051	5.1
				CLIP	436	31	109	48.6	191	857	4.49
squar5	5	8	56	*EXOC*	152	25	38	19.2	120	419	3.49
				DOM	140	25	35	4.4	120	419	3.49
				CLIP	152	25	38	4.5	121	426	3.52
xor5	5	1	32	*EXOC*	16	4	4	1.4	6	12	2
				DOM	16	4	4	0.4	6	12	2
				CLIP	16	4	4	0.4	6	12	2

Table 5.1: A Comparison of the results obtained by running *MVGUD* with 2 variables in the Bound Set on MCNC Benchmarks

we conclude that for two variables in the Bound Set it is not worthwhile having an Exact Graph Coloring to calculate the Column Multiplicity in *MVGUD*, and a good heuristic algorithm to calculate the Column Multiplicity is sufficient.

5.3.2.2 A Comparison when *MVGUD* is run with 4 and 5 variables in the Bound Set

As the next step *MVGUD* was run with 4 variables and with 5 variables in the Bound Set. The following tables show the results.

Comparison of *EXOC*, *DOM* and *CLIP* on MCNC Benchmarks											
Benchmark	in	out	cubes	option	DFC	nu of blocks	Av Edge%	T(s) (sec)	NP	TC	AC
5xp1	7	10	143	*EXOC*	344	17	62.75	2006	28	123	4.4
				CLIP	344	17		29.5	28	123	4.4
				DOM	344	17		29.9	28	123	4.4
9sym1	9	1	158	*EXOC*	96	3	-	108	11	54	4.9
				CLIP	96	3		55.2	10	52	5.2
				DOM	64	3		47.3	11	54	4.9
b12	15	9	172	*EXOC*	284	25	14.54	87	130	389	3
				CLIP	284	25		57.1	132	387	2.9
				DOM	284	25		46.4	130	389	3
bw	5	28	97	*EXOC*	560	56	55	51	115	361	3.14
				CLIP	560	56		50.9	115	361	3.14
				DOM	560	56		48.7	115	361	3.14
ex5p	8	63	214	*EXOC*	-	-	-	-	-	-	-
				CLIP	2472	186		565	8	35	4.4
				DOM	2472	186		516	8	35	4.4
misex1	8	7	40	*EXOC*	400	19	60.47	318	589	1831	3.1
				CLIP	388	19		24.7	587	1816	3.1
				DOM	400	19		22.5	589	1831	3.1
rd53	5	3	63	*EXOC*	80	6	63.5	5.4	8	26	3.25
				CLIP	80	6		4.7	8	26	3.25
				DOM	80	6		4.9	8	26	3.25
rd73	7	3	274	*EXOC*	160	6	71.6	46.6	10	38	3.8
				CLIP	160	6		41.6	10	38	3.8
				DOM	160	6		47.2	10	38	3.8
rd84	8	4	515	*EXOC*	220	12	-	189	23	75	3.3
				CLIP	220	12		127	24	80	3.3
				DOM	208	12		131	24	75	3.3
sao2	10	4	133	*EXOC*	416	12	67	423.8	52	337	6.5
				CLIP	416	12		124	52	337	6.5
				DOM	416	12		121	52	337	6.5
squar5	5	8	56	*EXOC*	200	16	58.6	124.8	31	94	3
				CLIP	200	16		10.2	31	94	3
				DOM	200	16		10.5	31	94	3
xor5	5	1	32	*EXOC*	20	2	53	1.1	2	4	2
				CLIP	20	2		1.2	2	4	2
				DOM	20	2		1.0	2	4	2

Table 5.2: A Comparison of results obtained by running *MVGUD* with 4 variables in the Bound Set on MCNC Benchmarks

Comparison of *EXOC*, *DOM* and *CLIP* on MCNC Benchmarks											
Benchmark	in	out	cubes	option	DFC	nu of blocks	Av Edge%	T(s) (sec)	NP	TC	AC
5xp1	7	10	143	*EXOC*	368	18	58.78	3240	37	136	3.7
				CLIP	368	18		64	37	136	3.7
				DOM	368	18		41.8	37	136	3.7
9sym	9	1	158	*EXOC*	128	2	77	125	5	30	6
				CLIP	128	2		54.6	5	30	6
				DOM	128	2		47.7	5	30	6
b12	15	9	172	*EXOC*	524	23	46.1	383	99	310	3.13
				CLIP	524	23		232	100	307	3.07
				DOM	524	23		89	98	307	3.13
bw	5	28	97	*EXOC*	896	28	55	29.1	-	-	-
				CLIP	896	28		29.2	-	-	-
				DOM	896	28		29.8	-	-	-
ex5p	8	63	214	*EXOC*	-	-	-	-	-	-	-
				CLIP	3520	124		1280.4	365	1258	3.45
				DOM	3520	124		724	365	1258	3.45
misex1	8	7	40	*EXOC*	432	14	56	4400	35	132	3.77
				CLIP	432	14		45.7	35	132	3.77
				DOM	432	14		23.6	35	132	3.77
misex2	25	18	101	*EXOC*	-	-	-	-	-	-	-
				CLIP	1152	43		26665	-	-	-
				DOM	1152	43		27540	-	-	-
sqrt8	8	4	66	*EXOC*	-	-	-	-	-	-	-
				CLIP	200	7		31	-	-	-
				DOM	200	7		21.3	-	-	-
inc	7	9	94	*EXOC*	480	18	58.44	28620	-	-	-
				CLIP	480	18		69	-	-	-
				DOM	480	18		46.9	-	-	-
con1	7	2	18	*EXOC*	80	4	71	1800	8	36	4.5
				CLIP	80	4		6.5	8	36	4.5
				DOM	80	4		4	8	36	4.5
rd73	7	3	274	*EXOC*	200	6	70	57.8	10	36	3.6
				CLIP	200	6		51.3	10	36	3.6
				DOM	200	6		49.6	10	36	3.6
rd84	8	4	515	*EXOC*	288	8	62.2	194	14	54	3.86
				CLIP	288	8		158	14	54	3.86
				DOM	288	8		159	14	54	3.86
sao2	10	4	133	*EXOC*	596	12	52	3240	41	289	7.05
				CLIP	596	12		231.4	43	287	6.7
				DOM	596	12		219	41	289	7.65
squar5	5	8	56	*EXOC*	256	8	58.6	5.6	-	-	-
				CLIP	256	8		5.6	-	-	-
				DOM	256	8		5.8	-	-	-
xor5	5	1	32	*EXOC*	32	1	53	0.5	-	-	-
				CLIP	32	1		0.6	-	-	-
				DOM	32	1		0.5	-	-	-

Table 5.3: A Comparison of results obtained by running *MVGUD* with 5 variables in the Bound Set on MCNC Benchmarks

In Tables 5.2 and 5.3 *Av Edge%* was calculated to see how dense or sparse the graphs generated during the decomposition are. This was calculated in the following way: For any graph with number of nodes = n, the *total_possible_edges* for

this graph(100% edges) will be equal to $n*(n-1)/2$. Hence if the number of edges in the graph is equal to e, then the $edge\_percent = (e*100)/total\_possible\_edges$. This will give the *edge_percent* in a graph with n nodes and e edges. Since the decomposer calls the function to calculate the Column Multiplicity a number of times, the Av Edge% was calculated by adding the *edge_percent* for a graph each time the function to find Column Multiplicity was called, and then dividing this total by the number of times the function to calculate the Column Multiplicity was called.

Table 5.2 shows the comparison of running *MVGUD* with 4 variables in the Bound Set, and Table 5.3 shows the comparison of running *MVGUD* with 5 variables in the Bound Set. Looking at the results it is seen that *EXOC*, *DOM* and *CLIP* generate the same results in all the cases in terms of DFC and number of CLB's. The reason for the slow times of *MVGUD* with *EXOC* can be explained as follows:When *MVGUD* is run with 5 variables in the Bound set, in most cases the average number of nodes in the graph is 32 and the edge percentage is always high with the highest being 77% and the lowest being 46.1% as shown in Table 5.3. This means that the graphs generated during decomposition were nearly always (since this is an average) dense graphs. Looking back at Tables 4.2 and 4.3 we see that for these dense graphs *EXOC* takes a long time to find the Exact solution, hence we have such slow times for *EXOC*. Also from Tables 4.2 and 4.3 we see that whenever *DOM* does not generate an exact solution, it is usually 1 or 2 colors away from the Exact solution and rarely more than that, and this being on randomly generated graphs. Now considering that there were 5 variables in the Bound Set, then the Incompatibility graph will have 32 nodes, and for a Curtis decomposition to exist, if a coloring of the graph with 16 colors or less is found then one exists. Now looking at Table 5.3 in the column for Average colors *AC* it can be seen that the largest average color is 7.65 for the benchmark *sao2* But this means that these graphs generated during decomposition, had low chromatic numbers, which were much less than 16. So even if *DOM* or *CLIP* generate a solution that is 2 or 3 colors away, the solution will be accepted as a Curtis De-

composition because it will still be less than 16. The same reasoning applies to Table 5.2 where a comparison is made with 4 variables in the Bound Set.
Hence we conclude that for 4 or 5 or greater number of variables in the Bound Set an Exact Graph Coloring does not produce better Curtis Decompositions, and having a good heuristic algorithm to find the Column Multiplicity or even a greedy algorithm to find the Column Multiplicity is good enough.

5.3.2.3 A Summary of the Results obtained by testing *DOM*, *EXOC* and *CLIP* on MCNC Benchmarks

Tables 5.1, 5.2, and 5.3 showed that having an Exact Method of calculating the Column Multiplicity in Functional Decomposition of circuit benchmarks is not necessary. In order to see why the results obtained for *MVGUD* with different algorithms for calculating the Column Multiplicity were so similar, *MVGUD* was tested to calculate the size of the difference in the Total number of Colors generated during the entire decomposition by *EXOC*, *CLIP* and *DOM*. But what was found was that, the algorithms *EXOC*, *CLIP* and *DOM* are at some stage of the Decomposition, choosing different Bound Sets as an acceptable decomposition. This is because they are different algorithms and so they are generating different colorings of the incompatibility graphs generated during the decomposition process. Hence this results in different "G" and "H" functions, when *MVGUD* is run with *EXOC*, with *CLIP*, or with *DOM*. This can be seen from the Column "NP" in Tables 5.1, 5.2, and 5.3 which show that depending on which algorithm *EXOC*, *CLIP* or *DOM* is used in *MVGUD*, the number of times the function to calculate the Column Multiplicity is called varies, which means that the Total Colors ("TC") varies, and thus cannot be compared. Hence to conclude this section, what we have proved here is that having an Exact method of calculating the Column Multiplicity for Circuit Benchmarks is not worthwhile in *MVGUD* for Bound Sets up-to five. Comparisons for Bound Sets greater than five has not been done, because then *EXOC* is too slow to generate results, for the large graphs involved here, hence not practical.

5.4 A Comparison of the different Strategies of finding the Column Multiplicity on Machine Learning Benchmarks

To see if the same results were obtained on Machine Learning Benchmarks, *MVGUD* was also tested on Machine Learning Benchmarks. These Benchmarks were obtained from the Wright Labs Database. These are completely specified functions with 8 and 12 variables in the input and one output. Since we were interested in don't-cares in the function the program *FLASH*(introduced in Chapter 1) was used to convert the above functions into functions with 70% of don't cares. The following Tables illustrate the results obtained. *MVGUD* was run in the same way as before, with 2, 4 and 5 variables in the Bound Set.

1. Tables 5.4, 5.5, and 5.6 show the results of running *MVGUD* on Machine Learning Benchmarks(MLB). These MLB have 8 variables in the Bound Set. In these tables they have 70% of don't cares. (cubes = 30% * 256)

2. Table 5.4 is a result of running *MVGUD* with 2 variables in the Bound Set.

3. Table 5.5 is a result of running *MVGUD* with 4 variables in the Bound Set.

4. Table 5.6 is a result of running *MVGUD* with 5 variables in the Bound Set.

5. On examining these Tables it was seen that here too *EXOC* fails to improve the quality of Decomposition. The results with all three algorithms prove to be nearly the same, with slight differences in some cases.

6. In Table 5.7, *MVGUD* was tested on the MLB, 12 variable functions, with the functions being completely specified, and with 70% of don't cares, and here too there is no change in the results. In Table 5.7 the functions having cubes = 1228 are fully specified and functions with 410 cubes are with 70% don't cares.

7. Testing was not done on Bound Sets greater than 5 because for Bound Sets greater than 5, *EXOC* is too slow to be practical.

Comparison of *EXOC*, *DOM* and *CLIP* on Machine Learning Benchmarks										
Function		Algorithm								
		CLIP			*EXOC*			*DOM*		
name	cubes	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)
add0	77	28	5	12.5	28	5	24.7	28	5	12.8
add2	77	28	6	12.9	28	5	13.1	28	5	13.1
add4	77	10	3	7.3	10	3	7.1	10	3	7.6
and_or_chain8	77	22	6	12	22	6	11.9	22	6	12.5
ch15f0	77	56	6	20.4	68	6	417	68	6	24.6
ch176f0	77	14	4	11.7	22	6	13	22	6	14
ch177f0	77	6	2	6.3	6	2	6.3	6	2	6.7
ch22f0	77	28	5	12	28	5	12.3	28	5	13.4
ch30f0	77	40	5	16.4	40	5	15.8	40	5	17.1
ch47f0	77	28	5	14.3	28	5	15	28	5	16.4
ch52f4	77	64	5	18.5	76	5	19.8	76	5	20.4
ch70f3	77	28	6	14.6	28	6	14.6	28	6	15.8
ch74f1	77	52	6	15	44	5	19.5	52	6	15.5
ch83f2	77	64	5	19.5	64	4	25.7	64	4	22.6
ch8f0	77	22	5	13.8	22	5	13.8	22	5	15.6
contains_4_one	77	28	5	19.5	44	4	23.1	44	4	24.1
greater_than	77	28	7	13.3	28	7	13.1	28	7	14.6
interval1	77	44	4	20.5	56	5	20.3	56	5	21.9
interval2	77	64	4	17.2	64	4	21.7	64	4	18.2
kdd1	77	22	6	8.9	16	5	8.4	16	5	9.7
kdd2	77	22	6	12.9	22	6	12.3	22	6	14
kdd3	77	14	4	7.8	14	4	7.2	14	4	8.9
kdd4	77	2	1	6.4	2	1	5.9	2	1	6.9
kdd5	77	28	6	12.3	28	6	13.5	28	6	14.4
kdd6	77	14	4	8.3	14	4	8.3	14	4	9.2
kdd7	77	28	7	10.5	28	5	14.2	28	5	15.4
kdd8	77	14	4	7.1	14	4	6.5	14	4	7.6
kdd9	77	28	6	11.7	28	7	11.7	28	7	12.9
majority_gate	77	28	6	14.5	40	5	16.6	40	5	17.2
modulus2	77	28	6	17	28	5	17.3	28	5	18.8
monkish1	77	14	4	8.2	14	4	8.3	14	4	9
monkish2	77	28	6	13.9	28	6	13.6	28	6	15
monkish3	77	22	5	9.7	22	5	9.5	22	5	10.7
mux8	77	46	5	12.8	46	5	12.2	46	5	13.8
nnr1	77	64	5	18.5	64	5	19.7	64	5	19.1
nnr2	77	14	3	6.7	14	3	6.7	14	3	7.8
nnr3	77	88	5	22.6	68	4	33.2	68	4	26
or_and_chain8	77	22	6	9	22	6	9.2	22	6	10.3
pal	77	22	5	13.9	22	5	13.4	22	5	14.6
pal_dbl_output	77	92	5	23.4	92	6	29.1	92	5	25
parity	77	28	7	11	28	7	11.3	28	7	12.8
primes8	77	68	5	23.3	68	4	26.4	68	4	24.6
remainder2	77	68	4	23.9	68	4	29.8	68	4	25.1
rnd1	77	152	5	23.5	152	5	110	152	5	23.2
rnd2	77	88	4	23	88	4	187	88	4	24.9
rnd3	77	88	5	20.7	88	5	33.8	88	4	26.1

Table 5.4: A Comparison of results obtained by running *MVGUD* with 2 variables in the Bound Set on 8 variable MLB

Comparison of *EXOC*, *DOM* and *CLIP* on Machine Learning Benchmarks										
Function		Algorithm								
		CLIP			*EXOC*			*DOM*		
name	cubes	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)
add0	77	36	3	11.8	36	3	9.8	36	3	9.7
add2	77	104	3	15.6	68	3	12.1	68	3	11.9
add4	77	18	2	9.5	18	2	8	18	2	8
and_or_chain8	77	36	3	10.6	36	3	9.3	36	3	10.1
ch15f0	77	104	3	11.6	104	3	9.4	104	3	11.9
ch176f0	77	36	3	13	36	3	10.4	36	3	10.4
ch177f0	77	36	3	10.2	18	2	7.3	18	2	8.2
ch22f0	77	36	3	9.9	36	3	8.3	36	3	9.3
ch30f0	77	56	3	12.3	56	3	10.5	56	3	10.7
ch47f0	77	36	3	12.4	36	3	11	36	3	11.6
ch52f4	77	36	3	14.1	56	3	11.3	56	3	12.5
ch70f3	77	36	3	14.3	36	3	11.4	36	3	10.7
ch74f1	77	36	3	13.9	36	3	10.9	36	3	10.2
ch83f2	77	56	3	14.8	56	3	12	56	3	12
ch8f0	77	36	3	12.5	36	3	10.9	36	3	10.6
contains_4_one	77	36	3	12.8	36	3	11.2	36	3	11.3
greater_than	77	36	3	12.1	36	3	9.3	36	3	10.1
interval1	77	36	3	13.3	36	3	11.4	36	3	11.1
interval2	77	80	3	11.9	80	3	10.5	80	3	9.6
kdd1	77	12	2	10.9	12	2	8.8	12	2	8.8
kdd2	77	36	3	12.7	36	3	10	36	3	10.5
kdd3	77	36	3	11.6	36	3	9.7	36	3	10.1
kdd4	77	12	2	5.9	12	2	4.9	12	2	4.6
kdd5	77	36	3	10.8	36	3	10.2	36	3	11.2
kdd6	77	18	2	7.4	18	2	6	18	2	6
kdd7	77	36	3	10.1	36	3	8.1	36	3	8.8
kdd8	77	12	2	6.3	12	2	5.2	12	2	5.2
kdd9	77	36	3	11.1	36	3	9.2	36	3	9.5
majority_gate	77	36	3	12.5	36	3	10.8	36	3	10.6
modulus2	77	36	3	14	36	3	10.5	36	3	10.7
monkish1	77	18	2	8.7	18	2	7.6	18	2	7.8
monkish2	77	36	3	12.8	36	3	10.4	36	3	9.9
monkish3	77	36	3	11.6	36	3	9.9	36	3	10.2
mux8	77	36	3	12.7	36	3	11	36	3	10.1
nnr1	77	36	3	11	36	3	8.7	36	3	8.8
nnr2	77	18	2	6.3	18	2	5	18	2	5.2
nnr3	77	68	3	15.7	68	3	11.9	68	3	11.8
or_and_chain8	77	36	3	10.4	36	3	9.3	36	3	9.3
pal	77	36	3	13.4	36	3	10.9	36	3	10.6
parity	77	36	3	10.8	36	3	8.9	36	3	9.2
primes8	77	68	3	13.8	68	3	12.1	68	3	11.9
remainder2	77	104	3	12.4	104	3	9.9	104	3	9.8
rnd1	77	104	3	16.1	104	3	236	104	3	11.3
rnd2	77	104	3	18	104	3	66	104	3	13.7
rnd3	77	104	3	16.3	104	3	660	104	3	12.6

Table 5.5: A Comparison of results obtained by running *MVGUD* with 4 variables in the Bound Set on 8 variable MLB

Comparison of *EXOC*, *DOM* and *CLIP* on Machine Learning Benchmarks										
Function		Algorithm								
		CLIP			*EXOC*			*DOM*		
name	cubes	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)
add0	77	48	2	13.3	48	2	9.3	48	2	9.1
add2	77	48	2	12	48	2	11.3	48	2	10.9
add4	77	48	2	12	48	2	12.1	48	2	11.5
and_or_chain8	77	48	2	21.7	48	2	13.4	48	2	11.9
ch15f0	77	96	2	15.4	96	2	11.4	96	2	11.1
ch176f0	77	48	2	21.7	48	2	12.3	48	2	12
ch177f0	77	48	2	14.2	48	2	9.6	48	2	10.2
ch22f0	77	48	2	17.1	48	2	10.8	48	2	10.6
ch30f0	77	48	2	19.4	48	2	11.3	48	2	12
ch47f0	77	96	2	16.9	96	2	13.3	96	2	13.1
ch52f4	77	96	2	21.4	96	2	13.5	96	2	12.4
ch70f3	77	48	2	20	48	2	13.1	48	2	13.2
ch74f1	77	48	2	19.2	48	2	12.7	48	2	11.7
ch83f2	77	96	2	17.4	96	2	16.6	96	2	13
ch8f0	77	48	2	21.7	48	2	12.1	48	2	12.5
contains_4_one	77	48	2	17.1	48	2	11.7	48	2	12.2
greater_than	77	48	2	17.1	48	2	11	48	2	10.7
interval1	77	48	2	17.4	48	2	11.5	48	2	10.9
interval2	77	96	2	16.1	96	2	12.1	96	2	14.3
kdd1	77	8	1	17.3	8	1	10.1	8	1	11.1
kdd2	77	48	2	18.4	48	2	10.4	48	2	11.2
kdd3	77	48	2	14.7	48	2	9.8	48	2	9.9
kdd4	77	8	1	8.7	8	1	6	8	1	5.4
kdd5	77	48	2	16.8	48	2	12.1	48	2	11.8
kdd6	77	48	2	21.9	48	2	12	48	2	13
kdd7	77	48	2	17	48	2	11.6	48	2	12.1
kdd8	77	48	2	18.3	48	2	11.5	48	2	12.6
kdd9	77	48	2	14.2	48	2	10.6	48	2	10.2
majority_gate	77	48	2	17.1	48	2	12	48	2	10.7
modulus2	77	48	2	15.1	48	2	10.8	48	2	10.8
monkish1	77	48	2	13.9	48	2	9.6	48	2	9.3
monkish2	77	48	2	15.7	48	2	12.5	48	2	13.1
monkish3	77	48	2	19.2	48	2	11.3	48	2	11.2
mux8	77	48	2	15	48	2	10.8	48	2	10.4
nnr1	77	48	2	16.4	48	2	11.9	48	2	12.4
nnr2	77	48	2	17.7	48	2	11.1	48	2	11.5
nnr3	77	96	2	17.5	96	2	13.9	96	2	12.4
or_and_chain8	77	48	2	18.9	48	2	12.1	48	2	12
pal	77	48	2	22.3	48	2	11.8	48	2	12.8
parity	77	48	2	16.4	48	2	11.1	48	2	11.5
primes8	77	96	2	19.4	96	2	13.7	96	2	13.3
remainder2	77	96	2	15.9	96	2	10.4	96	2	12
rnd1	77	96	2	17.6	96	2	12.8	96	2	13.8
rnd2	77	96	2	18.8	96	2	111	96	2	14.2
rnd3	77	96	2	17.8	96	2	14.2	96	2	12.8

Table 5.6: A Comparison of results obtained by running *MVGUD* with 5 variables in the Bound Set on 8 variable MLB

Comparison of *EXOC*, *DOM* and *CLIP* on Machine Learning Benchmarks										
Function		Algorithm								
		CLIP			*EXOC*			*DOM*		
name	cubes	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)
add0	1228	192	8	2100	192	8	2100	192	8	2160
add0_90	410	50	7	371	62	8	300	62	8	360
add2	1228	88	9	2040	88	9	2400	88	9	2100
add2_90	410	80	8	360	80	8	360	80	8	420
add4	1228	42	7	1080	42	7	1020	42	7	1140
add4_90	410	42	7	240	42	7	240	42	7	232
contains_4_one	1228	208	6	3600	208	6	7200	208	6	3600
contains_4_ones_90	410	164	7	600	164	7	1180	212	7	600
greater_than	1228	216	8	2760	216	8	4100	216	8	2820
greater_than_90	410	116	9	360	116	9	380	116	9	312
interval2	1228	60	7	3060	60	7	3500	60	7	3190
interval2_90	410	44	10	480	44	10	560	44	10	420
majority_gate	1228	208	6	3000	208	6	5200	208	6	2700
majority_gate_90	410	260	8	720	260	8	980	260	8	720
pal	1228	44	11	2640	44	11	3650 -	44	11	2640
pal_90	410	90	7	480	90	7	536	90	7	420
parity	1228	44	11	1620	44	11	1890	44	11	1680
parity_90	410	44	11	360	44	11	364	44	11	378
substr1	1228	140	9	2880	140	9	3240	140	9	2880
substr1_90	410	236	7	600	236	7	720	236	7	658
subtraction1	1228	120	8	3540	120	8	4890	120	8	3601
subtraction1_90	410	360	7	600	360	7	730	360	7	600
subtraction3	1228	316	10	4620	-	-	-	316	10	3615
subtraction3_90	410	312	9	600	312	9	612	312	9	600

Table 5.7: A Comparison of results obtained by running *MVGUD* with 2 variables in the Bound Set on 12 variable MLB

8. Testing was also done to compare the total count of colors generated during the process of decomposition, but here too it was found that the number of times the algorithms for calculating the column multiplicity is called varies for the same function. Hence these Tables are not included here.

5.4.1 A Summary of the Results Obtained from Testing on Machine Learning Benchmarks

As can be seen from Tables 5.4, 5.5, 5.6 and 5.7 *EXOC* was unable to provide a better *DFC* for the Machine Learning Benchmarks. In order to see the total numbers of colors generated by *DOM*, *EXOC* and *CLIP* on the same graphs, which were generated during the process of Functional Decomposition, the following experiment was performed: *MVGUD* was made to run with all three algorithms *EXOC*, *CLIP*, and *DOM* calculating the Column Multiplicity, and only the results of one of them was accepted and the results from the other two was discarded. The count of the colors was kept for all three Algorithms, thus demonstrating how *EXOC*, *CLIP*, and *DOM* compare with respect to the total number of colors generated on the same graphs, only now these graphs have been generated from practical function Benchmarks. Table 5.8 shows the result of this comparison. Table 5.8 shows the results of running *MVGUD* with all three algorithms, *DOM*, *EXOC*, and *CLIP* on the same graphs which were generated during decomposition. Table 5.9 is a summary of the results of Table 5.8. Table 5.9 shows how *DOM* and *CLIP* compare with respect to the number of times that the total number of colors generated by *DOM* and *CLIP* are the same as the total number of colors generated by *EXOC*, and the number of times the total colors generated by *DOM* and *CLIP* were not exact and by how much.

1. In Table 5.9, the row *Exact* stands for the case when the total numbers of colors generated by *DOM* and *CLIP* was the same as the total colors generated by *EXOC*. *Error 1* stands for the case in which the total numbers of colors generated by *DOM* and *CLIP* were one color away from the total

numbers of colors generated by *EXOC*, and so on, till *Error 6*. Corresponding to these rows, the column *Nu* gives the number of times, and column *%* is equal to $Nu/TotalNumberofProgramRuns * 100$.

2. As can be seen from the Table 5.9, *DOM* performs extremely well, and *CLIP* does not perform so well. *DOM* thus proves to be a very good heuristic algorithm.

3. Table 5.10 is a total of the rows of Table 5.9 for *DOM* and *CLIP*.

Comparison of the Total Colors generated on Graphs from Machine Learning 8 variable Benchmarks									
Benchmark	Total Colors								
	2 variables in Bound			4 variables in Bound			5 variables in Bound		
	DOM	*EXOC*	*CLIP*	*DOM*	*EXOC*	*CLIP*	*DOM*	*EXOC*	*CLIP*
add0	79	79	79	36	36	**38**	15	15	15
add2	90	90	90	**46**	45	**48**	21	21	**24**
add4	20	20	20	13	13	13	12	12	**14**
and_or_chain8	69	69	69	21	21	**22**	17	17	**18**
ch15f0	156	156	**158**	36	36	**38**	**27**	23	**24**
ch176f0	35	35	**36**	22	22	**23**	15	15	15
ch177f0	15	15	15	17	17	17	10	10	**11**
ch22f0	61	61	61	**18**	18	18	15	15	15
ch30f0	126	126	126	33	33	33	22	22	**23**
ch47f0	111	111	111	33	33	33	27	26	**27**
ch52f4	141	141	**142**	37	37	**40**	24	24	**25**
ch70f3	73	73	**73**	28	28	28	20	20	20
ch74f1	105	105	105	26	26	**29**	25	25	25
ch83f2	123	123	123	44	44	**45**	28	28	**29**
ch8f0	90	90	90	21	21	21	17	17	**18**
contains_4_ones	126	126	126	32	32	**34**	24	23	**25**
greater_than	100	100	100	35	35	35	20	20	**21**
interval1	134	134	**135**	31	31	**32**	**18**	17	**19**
interval2	123	123	**124**	35	35	**36**	30	30	**32**
kdd1	46	46	46	19	19	19	12	12	12
kdd2	58	58	58	18	18	18	12	12	12
kdd3	38	38	38	23	23	23	13	13	13
kdd4	16	16	16	7	7	7	3	3	3
kdd5	89	89	**89**	31	31	**32**	20	20	**24**
kdd6	29	29	29	9	9	9	12	12	12
kdd7	59	59	59	24	24	**25**	**21**	20	**21**
kdd8	30	30	30	11	11	11	13	13	**14**
kdd9	77	77	77	27	27	27	18	18	**20**
majority_gate	94	94	94	32	32	32	21	21	**23**
modulus2	118	118	**119**	36	36	**39**	20	20	**22**
monkish1	31	31	31	14	14	**15**	**12**	11	**13**
monkish2	**84**	**84**	**85**	**34**	**34**	**37**	23	23	**25**
monkish3	73	73	73	28	28	**29**	17	17	17
mux8	102	102	102	**38**	38	38	19	19	**22**
nnr1	131	131	**134**	27	27	**30**	26	26	**30**
nnr2	36	36	36	11	11	11	15	15	15
nnr3	187	187	**189**	42	42	**45**	31	31	**34**
or_and_chain8	56	56	56	24	24	24	15	15	**16**
pal	74	74	74	23	23	**24**	14	14	14
pal_dbl_output	219	219	**220**	43	43	**46**	**33**	31	**37**
parity	46	46	46	18	18	**19**	12	12	**14**
primes8	124	124	124	36	36	**37**	27	27	**28**
remainder2	157	157	**159**	34	34	**35**	24	24	**29**
rnd1	205	205	**207**	49	49	**50**	34	34	**39**
rnd2	215	215	**216**	54	54	**56**	37	37	**42**
rnd3	156	156	**160**	52	52	**54**	33	32	**34**

Table 5.8: A Comparison of the Total Colors obtained by running *MVGUD* on Machine Learning Benchmarks

Summary of Results of Comparison of the Total Colors												
Total Number of Program Runs = 46												
	DOM						*CLIP*					
	Bound = 2		Bound = 4		Bound = 5		Bound = 2		Bound = 4		Bound = 5	
	Nu	%	Nu	%	Nu	%	Nu	%	Nu	%	Nu	%
Exact	46	100	45	97.8	41	89.1	32	66.1	20	43.5	14	30.5
Error 1	-	-	1	2.1	3	6.5	8	17.4	13	28.3	11	23.9
Error 2	-	-	-	-	1	2.1	4	8.6	5	10.8	12	26.1
Error 3	-	-	-	-	-	-	1	2.1	8	17.4	3	6.5
Error 4	-	-	-	-	1	2.1	1	2.1	-	-	3	6.5
Error 5	-	-	-	-	-	-	-	-	-	-	2	4.3
Error 6	-	-	-	-	-	-	-	-	-	-	1	2.1

Table 5.9: A Comparison of Total Colors generated by *DOM*, and *CLIP* compared with total colors generated by *EXOC* on the same graphs for two, four and five variables in the Bound Set for Machine Learning Benchmarks

Sum of Table 1				
Total Number of Program Runs = 138				
	DOM		*CLIP*	
	Total Number	Total %	Total Number	Total %
Exact	132	95.6	66	47.8
Error 1	4	2.8	33	23.9
Error 2	1	0.7	21	15.2
Error 3	-	-	12	8.7
Error 4	1	0.7	4	2.8
Error 5	-	-	2	1.4
Error 6	-	-	1	0.7

Table 5.10: A Summary showing the Addition of the Total Colors obtained in Table 5.9

5.5 Some Questions and Conclusions Reached from the Results of the Testing

We believe that to solve any problem it is important to go deep into the problem and investigate the problem carefully. Here the problem for us was Functional Decomposition. Functional Decomposition has always been a very complex problem, and all the research done till date just tries to solve the problem more efficiently without trying to reason why Functional Decomposition is such a difficult problem. Here we have investigated only the part of Functional Decomposition which involves finding the Column Multiplicity, but the results obtained in this Chapter provide a very deep insight into the Column Multiplicity part of Functional Decomposition. By the results of the testing we can definitely say that we have proved that an Exact Graph Coloring is not required to find the Column Multiplicity where Curtis Decompositions are considered. Exact Graph Colorings only take up more time and fail to produce any significant change in the results. This is true with respect to both Circuit Benchmarks and Machine Learning Benchmarks.

Also the results shown in the Summary Tables 5.9 and 5.10 raise the question that in cases where *CLIP* did not generate the same total numbers of colors as *EXOC*, why did the *DFC* not improve when we used *EXOC*? The only possible answer to this question is that the decompositions generated by *CLIP* were still acceptable decompositions, even if they use non minimum numbers of colors which in turn means that these graphs generated during the decomposition process must be having low chromatic numbers. This provides a very valuable insight into the kinds of graphs that are generated during the decomposition process. This tells us that the graphs generated during the decomposition process are definitely of a different nature than random graphs. Which raises another question: Why are the graphs generated during the decomposition process different from the graphs generated randomly? This question is beyond the scope of this thesis, but it is a very interesting question, because answering this question

might provide a better insight into the Functional Decomposition process.

5.6 What is the next step to solve the Column Multiplicity Problem

In this Chapter the heuristic graph coloring program (*DOM*), the exact graph coloring program (*EXOC*) and the clique partitioning program (*CLIP*) were tested in decomposition and the results proved that an Exact Graph Coloring does not yield better Curtis Decompositions. Now we know that there are basically four methods to find the Column Multiplicity, namely: Clique Partitioning, Clique Covering, Graph Coloring and Set Covering. Out of these methods the Clique Partitioning and Graph Coloring have been investigated by us. Hence what was needed now was to try either a Clique Covering or a Set Covering to see if they can improve the quality of decompositions achieved. Clique Covering and Set Covering are both similar, a Clique Covering is done on a compatibility graph while a Set Covering is done on a table. Exactly analogous to a Clique Covering is a Multi-Coloring. The only difference being that a Multi-Coloring is done on an incompatibility graph. Hence it was needed to test one of these methods in the Decomposition process. Since we had already written two Graph Coloring programs, it was decided to write a Multi-Coloring program in order to test the Column Multiplicity problem in Functional Decomposition further. Multi-Coloring, and the relation between a Multi-Coloring and a Clique-Covering is explained in more detail in the next Chapter.

CHAPTER 6

A NEW APPROACH TO MULTI-COLORING USING DOMINATIONS: COLUMN COMPATIBILITY FOR FUNCTIONS AND RELATIONS

This Chapter begins with an introduction of some basic Notions and Definitions, because before the different sections in this Chapter are introduced, it is necessary to understand these notions and definitions.

6.1 Basic Notions and Definitions

6.1.1 Multi-Coloring

Definition 6.1 *A multi-coloring of a graph is assigning a node with as many colors as possible, without creating a conflict between any two nodes, meaning, the sets of colors assigned to any two adjacent nodes are disjoint.*

6.1.2 Maximum Independent Sets in an Incompatibility Graph

Definition 6.2 *A Maximum Independent Set in an incompatibility graph is a maximum set of nodes in the incompatibility graph which can be given the same color.*

6.2 A Comparison showing the Equivalence of a Multi-Coloring and a Maximum Clique Covering

Figure 6.1 shows a comparison between a Maximum Clique Covering and a Multi-Coloring. Figure 6.1(a) is the incompatibility Graph with the multi-colors assigned. Figure 6.1(b) shows the same realization of the function, only now it is a

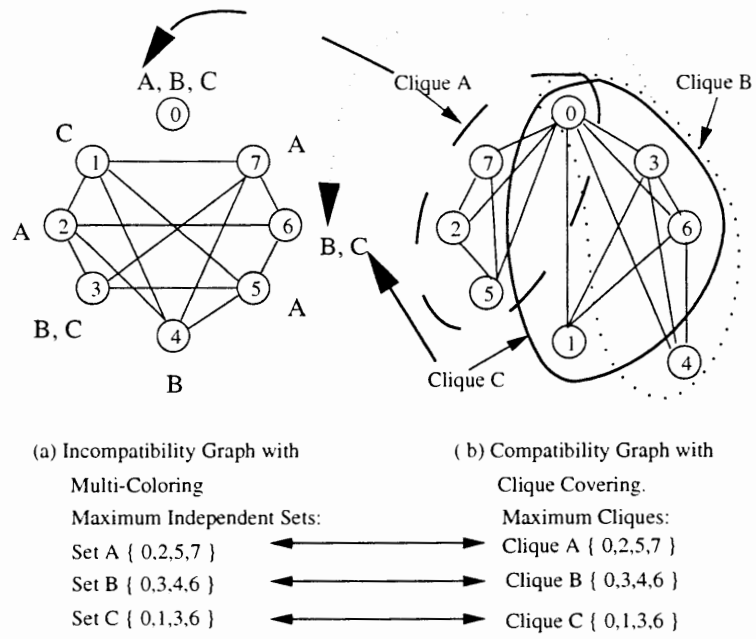


Figure 6.1: Difference between Clique Covering and Multi-Coloring

Compatibility Graph. From Figure 6.1(b) it can be seen that the Clique Covering produced 3 cliques. The same information was obtained from the multi-coloring, shown in Figure 6.1(a). Also it can be seen from Figure 6.1, that in Figure 6.1(a) the Maximum Independent Set A in the incompatibility graph is the same as Clique A in the compatibility graph in Figure 6.1(b). Similarly, Maximum Independent Set B is the same as Clique B, and the Maximum Independent Set C is the same as Clique C.

6.3 Why it was decided to use a Multi-Coloring in Decomposition

This thesis started of with the testing of the Column Multiplicity problem in Functional Decomposition, and in Chapter 5, it was proved that having an Exact Graph Coloring to solve the problem of Column Multiplicity in Functional Decomposition does not improve the DFC. Hence this meant that maybe what is needed is a Multi-Coloring (A Multi-Coloring is functionally equivalent to a Clique Covering as shown in Section 6.2), to solve the problem of Column Multi-

plicity in Functional Decomposition. Now as can be seen from Section 6.1.1, for a Multi-Coloring or Clique-Covering to improve the quality of decompositions achieved, the decomposer must make use of the overlapping independent sets. At the same time Dr Perkowski developed the new concept of Relations, and *MVGUD* was extended so that it now had the ability to decompose Relations too. This new concept of Relations provided the ability to take advantage of the overlaps obtained by the Multi-Coloring, thus with Relations the Multi-Coloring could be properly tested and evaluated. In the next Section Multi-Valued Relations and their applications will be introduced. This is the application in which we will show how the Multi-Coloring can improve the quality of the Curtis decompositions achieved. Then the details of the Multi- Coloring program are introduced. In the next Chapter an evaluation of how the Multi-Coloring performs on random graphs is presented. An Analysis of the Multi-Coloring on graphs generated during the decomposition process is also given in the next Chapter.

6.3.1 An Introduction to Multi-Valued Relations

Until very recently, logic synthesis focussed on efficient methods and algorithms to optimize the hardware of digital computers and other digital circuits. In the last few years an increased trend has occurred to apply the logic synthesis methods also in image processing, machine learning, knowledge discovery, database optimization, AI, image coding, automatic theorem proving and verification of software and hardware. What was found was that these Machine Learning Functions have quite different properties than Circuit Functions(MCNC). The characteristics of the Machine Learning data is:

1. Machine Learning Functions have a very high number of don't cares, typically more than 99%.

   Definition 6.3 *A don't care is defined as a combination of argument values for which the function value is not specified.*

In the Machine Learning community don't cares are referred to as unknown.

2. Most Machine Learning Functions are Multi-Output.

3. Practical Machine Learning problems require at least 30 binary input variables but more typically, about 100 multi-valued variables. As the number of input and output variables increases there is only a small increase in the number of positive and negative samples but a dramatic increase in the number of don't cares.

4. Arguments (variables) in Machine Learning problems are naturally multiple-valued (or continuous and then discretized).

5. In the context of Machine Learning when one talks of "noise", one means a situation when there is some data that was classified correctly, but then noise causes it to change its value. Another situation is when the value of a minterm is or becomes a don't care. This would be referred to as an unknown in the Machine Learning community. This is however not a very good idea and some information is lost in the preprocessing of the data for the Machine Learning program. In [22, 23, 24, 25] a more general solution has been proposed. Let us assume that the choices for a three valued decision variable X are: "a car", "a tank", and "an airplane", denoted as values X^0, X^1, and X^2 respectively. Then a standard don't care, I do not know, would correspond to $X^{0,1,2}$, denoted in a standard way by "-". But a decision saying "I do not know but not an airplane" would be a relation denoted by $X^{0,1}$. This is called a *generalized don't care*. If a function has generalized don't cares in the output then it is no longer called a function, now it is called a *relation*.

Table 6.1 shows a multi-valued *relation* with four binary inputs and one five valued output. In a *function* the output can only have one value, but as shown in Table 6.1 in row 0, minterm 0000 maps to the output $f = 0$ or 1. Therefore this row specifies a generalized don't care. This says that for input minterm 0000

row_no	a	b	c	d	f
0	0	0	0	0	0,1
1	0	1	0	0	1,2
2	1	1	0	0	0
3	0	0	0	1	0,3
4	1	1	0	1	0,3
5	1	0	0	1	0,4
6	1	0	0	0	0,3
7	0	0	1	1	1,3
8	0	1	1	1	0,1
9	0	0	1	0	2,3
10	1	0	1	0	1,4
11	0	1	1	0	2,3

Table 6.1: Table showing Relations in a four binary input and one multi-output Function

the output f can be a 0 or a 1 but not both. The designer has the freedom to select any one of these values, whichever simplifies the final description more. In this Table if we had a row with all five values $f^{0,1,2,3,4,5}$ = 0,1,2,3,4,5, then this would correspond to a classical don't care. If there is only a subset of values for example $f^{0,1}$, then this specifies a a generalized don't care. To better understand this assume that the meaning of the values of the output decision f stand for: 0 - a bicycle, 1 - a motorcycle, 2 - a car, 3 - a bus, 4 - a train, and 5 - an airplane. Then the position 0,1 in the output f in row 1 will stand for either a bicycle or a motorcycle, which means something is known but what is known is not precise. Here a value 0,1,2,3,4,5 would stand for a complete unknown. This corresponds to a standard don't care for functions. More details on relations and decomposition of Multi-Valued Relations can be obtained from [22, 23, 24, 25].

6.3.2 An Example showing Decomposition of Multiple-Valued Relations

In Figure 6.2 an example is shown which illustrates how to perform decompositions of Multi-Valued Relations. Figure 6.2(a) shows the Karnaugh map of a relation having four inputs and one 5 valued output. From this Karnaugh map a compatibility graph is created, where the nodes of the compatibility graph are a direct mapping of the columns of the Karnaugh map.

Definition 6.4 *Two columns of a Karnaugh map of a single output Multi-Valued Relation with generalized don't cares in the output are compatible if each entry in the two columns, has at least one value in common.*

Thus the compatibility graph is created. Figure 6.2(b) shows the compatibility graph. In Figure 6.2(b) the columns in brackets near the edges between two nodes C_i and C_j represents the combined column C_{ij}. One important point to be made here is: When there are generalized don't cares in the Karnaugh map, the checking of Compatibility of columns is different than the check for incomplete tautology defined earlier. This is because if Columns C0, C1, and C2 are pairwise compatible, it does not mean that they are compatible as a clique. From the compatibility graph the Cliques {B0, B1} and {B1, B2, B3} are obtained. The colors assigned are:

Column B0 : Color A,

Column B1 : Color A,B,

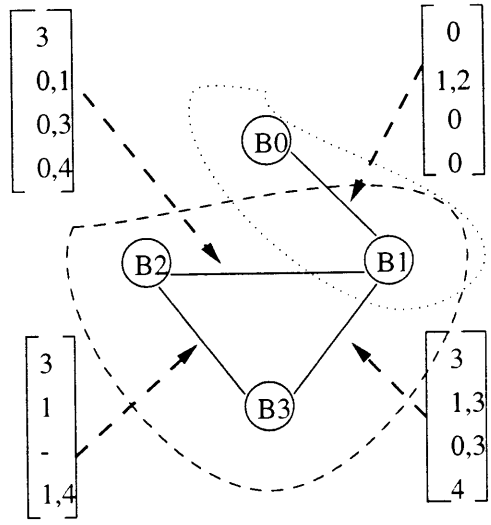
Column B2 : Color B,

Column B3 : Color B.

Then Column B0 is given the code $v = 0$, Columns B2 and B3 are given the code $v = 1$, and column B1 is given the code 0 and code 1. From this the maps of relations G and H are built. Thus this explains a disjoint Curtis decomposition of a relation F, into G and H relations.

ab \ cd	B0 00	B1 01	B2 11	B3 10
00	0,2	0,3	1,3	2,3
01	1,2	-	0,1	1,3
10	0	0,3	-	-
11	0,3	0,4	-	1,4

(a) Karnaugh map of Function F



(b) Compatibility graph of Function F.

c \ d	0	1
0	0	0,1
1	1	1

(c) G function with Relations

ab \ g	0	1
00	0	3
01	1,2	1
11	0	0,3
10	0	4

(d) H function with Relations

Figure 6.2: An Example showing Decomposition of Multiple-Valued Relations

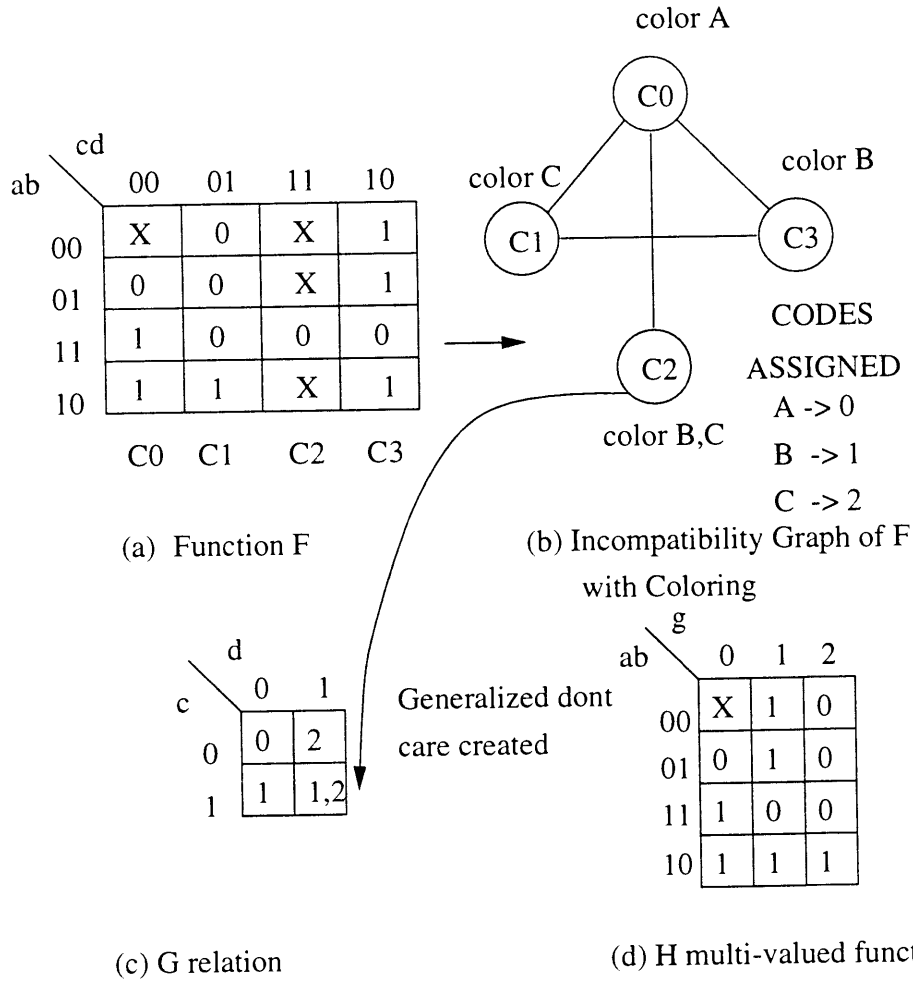
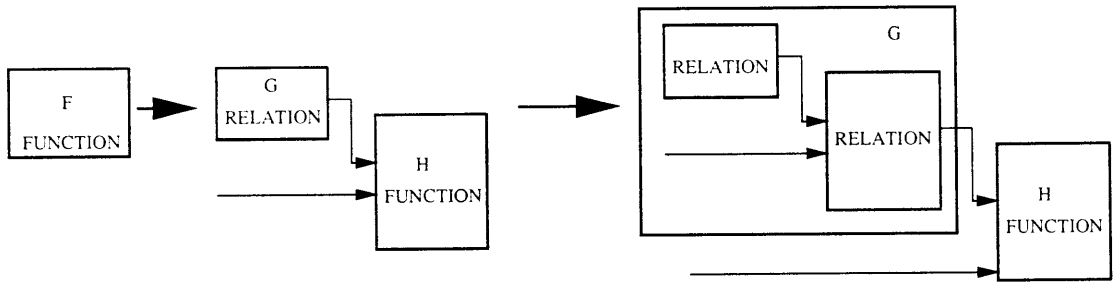


Figure 6.3: An Example showing Decomposition of a Binary Function and the Creation of Relations

6.3.3 An Example showing Decomposition of a Binary Function and the creation of Relations

In the example shown in Figure 6.3 it is shown how relations can be created from a Function by using a Multi-Coloring program to find the column multiplicity. Figure 6.3(a) shows a function with four inputs and one output. Figure 6.3(b) shows the incompatibility graph created from the Karnaugh map. Since this is a function, the checking of the compatibility of columns is a check for an incomplete tautology. On doing a multi-coloring of the incompatibility graph it is found that



(a) Initial Function F

(b) Initial Step of Decomposition

(c) Second Step of Decomposition

Figure 6.4: Decomposing a Function and using a Multi-Coloring to produce G Relations and H functions

column C2 can have two possible colors. The final colors assigned are:

Color A: {C0},

Color B: {C2,C3}.

Color C: {C1,C2}.

Column C0 is given the code $v = 0$, column C1 is given the code $v = 2$, column C3 is given the code $v = 1$, and column C2 is given the codes $v = 1,2$. Thus as a result of creating the G block, a multi-valued generalized don't care is obtained, as shown in Figure 6.3(c). But since the H block is formed from the original Karnaugh map from Figure 6.3(a) (which is a function), the H block will be a function and not a relation. But as can be seen from Figure 6.3(d) now the H block is a multi-valued function. Since this H function will be used for the next level of decomposition, in the next level too the H block cannot be a relation. From the examples shown in 6.3.2 and 6.3.3 the following theorems can be formulated:

Theorem 6.1 *If a* **function F** *is being decomposed, and if a Multi-Coloring is used to find the column multiplicity the G block can be a* **relation** *but the H block will always be a* **function**.

Proof 6.1 *In order to perform a Curtis Decomposition of a* **function**, *the G block is created from the encodings given to the intermediate variables. Thus if*

a Multi-Coloring is used to find the column multiplicity, then a column can have more than one code, which will result in generalized don't cares in the G block, thus making the block G a relation. But the H block is created directly from the columns of the Karnaugh map of the original function(explained in Chapter 2, Section 2.4), and since the original input is a function, the H block cannot be a relation. Now as this new H block is used for the next step of the decomposition process, in the next step of decomposition too the H block will not be a relation. The only possible situation when the H block can be a **relation** *is if a decomposed G block which is a relation is decomposed further, but this means that the input is a* **relation**. *This proves Theorem 6.1.*

Figure 6.4 illustrates Theorem 6.1. Figure 6.4(a) shows a Function, which is decomposed using a Multi-Coloring to find the Column Multiplicity, resulting in a G relation and an H function in Figure 6.4(b). The H function in Figure 6.4(b) is decomposed next, so also in the next step of decomposition the G block is a relation but the H block is a function (as shown in Figure 6.4(c)).

Theorem 6.2 *If a* **relation F** *is being decomposed, the H block can be a* **relation**, *if a graph coloring is used then the G block will be a* **function**, *but if a Multi-Coloring is used then only the G block can be a* **relation**.

Proof 6.2 *Now the input is a* **relation**. *As the H block is created from the columns of the original Karnaugh map (explained in Chapter 2, Section 2.4), which in this case specifies a* **relation**, *if the combination of compatible columns in the Karnaugh map leads to a column with generalized don't cares, the H block will be a* **relation**. *When creating the G block the original Karnaugh map is not used, instead the encodings chosen for the intermediate variables are used to create the G block(explained in Chapter 2, Section 2.4). Thus if a column does not have more than one code, then the G block cannot have any generalized don't cares, and a column can have more than one code only if a Multi-Coloring or a Clique Covering is used to find the column multiplicity, thus proving Theorem 6.2.*

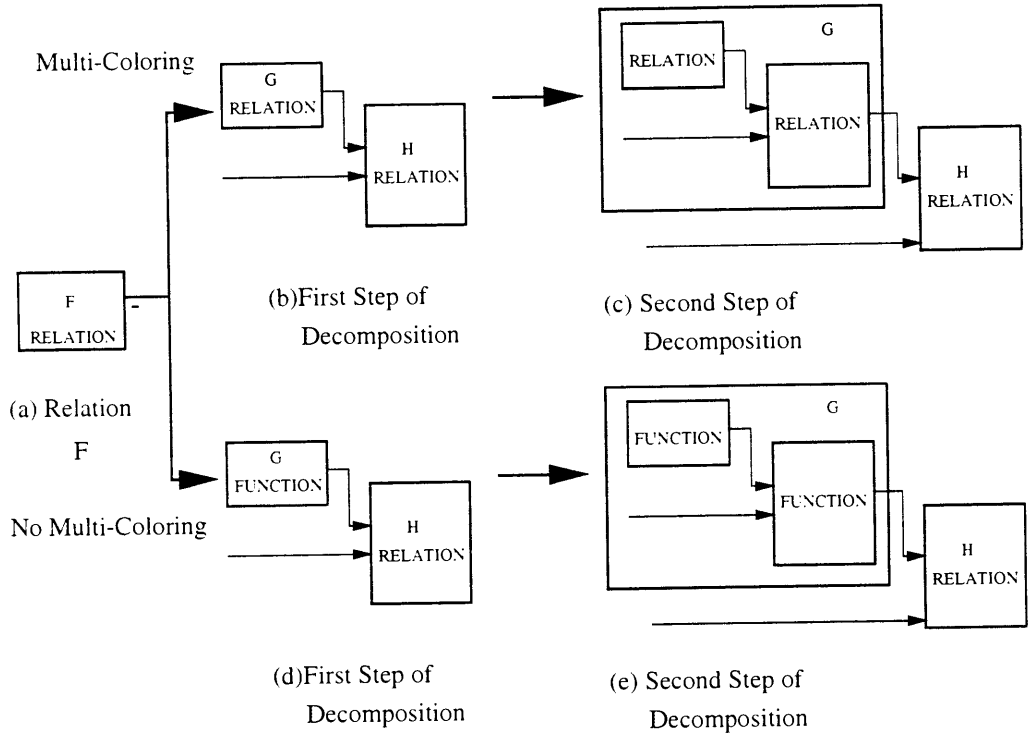


Figure 6.5: The Difference Decompositions obtained by using a Graph Coloring or a Multi-Coloring while Decomposing a Relation

Figure 6.5(a) shows a Relation. Figure 6.5(b) and Figure 6.5(c) shows the resulting decomposition if a Multi-Coloring is used to find the column multiplicity, and Figure 6.5(d) and Figure 6.5(e) present the resulting decomposition when a graph coloring is used. As can be seen, when a Multi-Coloring is used both the G and H blocks can be relations while when using a Graph Coloring only the H block can be a relation.

The advantage of having relations is that now the Functional Decomposer has the freedom to select any of the values specified for a generalized don't care, whichever simplifies the final description more. Thus the more the generalized don't cares in the relation, the more is the freedom to find the decomposition of the lowest cost.
Relations occur in various applications in Machine Learning, and Knowledge Discovery from Databases. Relations also occur in the areas of multi-level logic design and non-deterministic state machines. Hence Decomposition of Relations has applications in all these fields. Also with a Multi-Coloring we have the capability of creating relations even if the input data is a function. In the next section the Multi-Coloring algorithm will be presented and then it will be seen what kind of relations it can produce.

6.4 A New Approach to Multi-Coloring using Dominations

Here a new approach to Multi-Coloring is presented. The algorithm uses the same principle of dominations in an incompatibility graph, which was presented in Chapter 3. This algorithm has been given the name *MISDOM* which stands for *Maximum Independent Sets using Dominations.* In all latter sections, this algorithm will be referred to by this name. In the previous Chapter it was seen that an Exact Graph Coloring is not necessary to find the Column Multiplicity in Functional Decomposition, also from the results obtained in the previous Chapter, we saw that from a total number of program runs of 138, *DOM* gave the exact

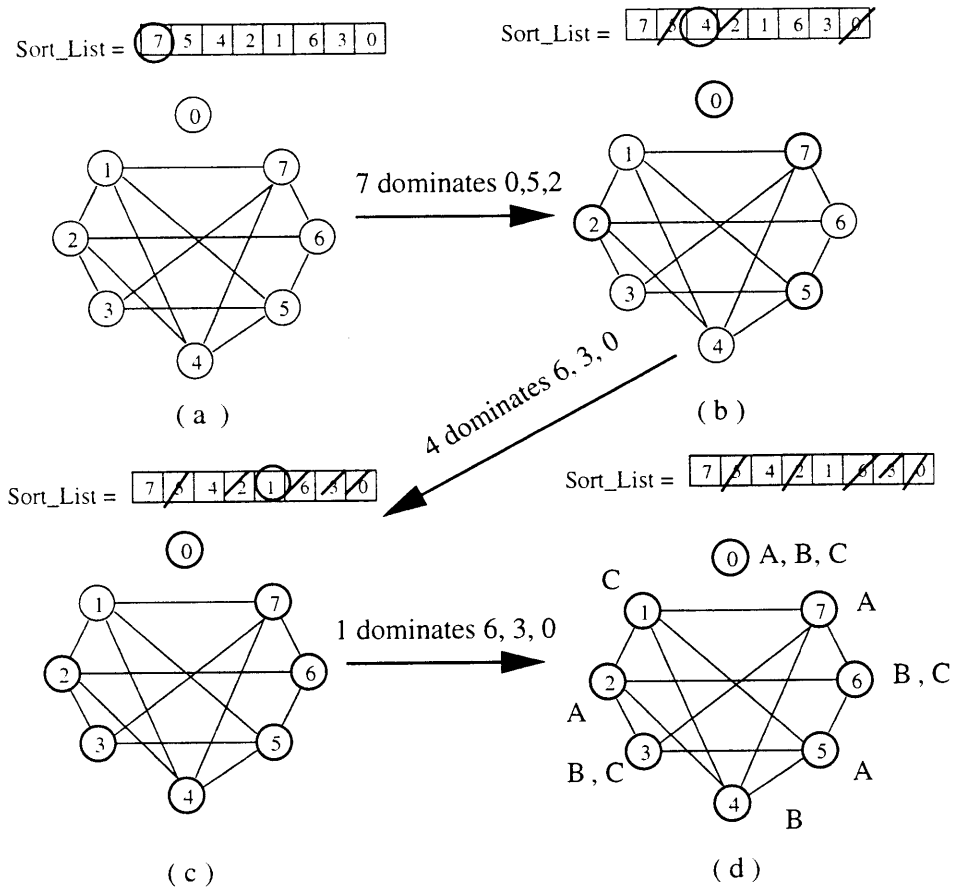


Figure 6.6: An Example showing how *MISDOM* colors a non cyclic Graph

solution in 95.6% of the cases, which proved that *DOM* is a very good heuristic. Hence we did not program a Multi-Coloring to be Exact. Instead it was decided to use an idea similar to *DOM*, thus ensuring that *MISDOM* will generate the same number of colors as *DOM* on any incompatibility graph, the only difference being, *MISDOM* is a Multi-Coloring. To explain how *MISDOM* works, first an example is presented, which shows how *MISDOM* colors a graph. Then the Algorithm and Pseudo Code for *MISDOM* will be presented.

6.4.1 Example showing how *MISDOM* colors a non cyclic Graph

Figure 6.6 shows an Example illustrating how the multi-coloring is done by *MISDOM*.

1. First the nodes of the incompatibility graph are sorted in a descending order of the vertex degrees, and a $SORT\_LIST$ is generated. The first element of $SORT\_LIST$ will have the node with the highest degree, and the last element of $SORT\_LIST$ will have the node with the smallest degree. The length of $SORT\_LIST$ will be equal to the number of nodes in the incompatibility graph. In Figure 6.6(a) the initial $SORT\_LIST$ and the initial Incompatibility Graph can be seen.
   Select first node from $SORT\_LIST$. Node 7 in this Example.

2. As can be seen from Figure 6.6(a) Node 7 dominates Node 0 and pseudo dominates Node 5, and Node 2. This information is stored. Remove Node 0, Node 5, and Node 2 from the $SORT\_LIST$, this is shown in Figure 6.6(b). Now select the next node in $SORT\_LIST$ which is Node 4.

3. In Figure 6.6(c) it can be seen that Node 4 dominates Node 6, Node 3 and Node 0. This information is stored, and Node 6, Node 3 and Node 0 are removed from the $SORT\_LIST$ if they have not already been removed. This is shown in Figure 6.6(c). Now take next node in $SORT\_LIST$ which is Node 1.

4. In the graph of Figure 6.6(d), Node 1 dominates Node 6, Node 3, and Node 0. This information is stored, and Node 6, Node 3 and Node 0 are removed from the $SORT\_LIST$ if they have not already been removed. Now the end of $SORT\_LIST$ has been reached so the check for dominations terminates.

5. Now each Dominating Node is given a new color, and all the nodes it dominates are given the same color. If a node is dominated by more than one node it is given the colors of all the nodes that dominate it.

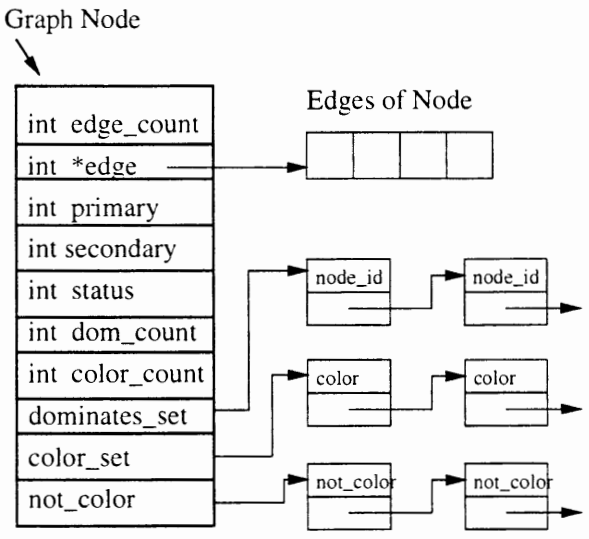


Figure 6.7: Data Structure used for a Node of the Incompatibility Graph

6. The final colored graph is shown in Figure 6.6(d). 3 colors were used to color the graph which is the minimum for this Graph. The color assignment is:

   Color A {0, 2, 5, 7}.

   Color B {0, 3, 4, 6}.

   Color C {0, 1, 3, 6}.

6.4.2 Implementation Details for *MISDOM*

Before going into the details of the Algorithm used by *MISDOM* a brief introduction to the data structures used by *MISDOM* is given below.

Figure 6.7 shows the data structure used to implement a Node in the incompatibility graph. In the graph node data structure shown in Figure 6.7, **edge_count** is the number of neighbors that the node has. **edge** is a pointer to the nodes that are the neighbors of the node. **primary** indicates that it is a primary node if set to "1". **secondary** indicates that it is a secondary node if set to "1". **status** is set to "1" if the node has been assigned a color, else it is set to "0". **dom_count** is a count of the number of nodes that are dominated by

this node. **color_count** is the number of colors that have been assigned to the node. **dominates_set** is a pointer to a link list which indicates which nodes are dominated by this node. **color_set** is a pointer to a link list containing all the colors that have been assigned to the node. **not_color** is a link list containing the colors that the node cannot be colored.

6.4.3 Details of the Multi-Coloring Program, *MISDOM*

The Algorithm for *MISDOM* has been divided into two parts:

1. Function *check_dominations*, which checks for dominations in the graph. It has two possible return values.

   (i) Returns "TRUE" if graph was complete.

   (ii) Returns "FALSE" if graph was cyclic, or disjoint.

2. Function *graph_coloring* is the top level function, which calls *check_dominations*.

The Algorithm for the Function *graph_coloring* has not been given here because it is exactly the same as the algorithm explained in Chapter 3 for *DOM*. The algorithm for the Function *check_dominations* is different and has been explained here.

6.4.3.1 Algorithm for Function *check_dominations*

$NODES$	:is the set of nodes.
$NODE\_COUNT$	:is the number of nodes.
$SORT\_LIST$	:is an array of sorted nodes. The first element in array $SORT\_LIST$ is the node with the most number of neighbors, and the last element is the node with the least number of neighbors. The length of $SORT\_LIST$ is equal to the number of nodes in the

incompatibility graph.

1. Sort the nodes as explained in Step 1 of the example in Section 6.4.1 and generate a list $SORT\_LIST$. Mark all nodes as being *secondary* initially (set secondary in Figure 6.7 to "0").

2. Take one node from $SORT\_LIST$, selected node, $SN = SORT\_LIST$[i]. Mark this node as *primary* in the node data structure for this node.

3. Take one node from nodes ($NODE = NODES[i]$).
   /** Only check for dominations if $NODE$ is a secondary node. All the information about the node is obtained from the data structure of the node shown in Figure 6.7. If a domination is found, the domination information is stored in the graph node of the dominating node. **/
   If ($NODE == secondary$)
   Check for Dominations of (SN, $NODE$);
   If Domination found, mark Dominated node as being dominated by SN. Do not remove Dominated node from graph, instead remove dominated node from $SORT\_LIST$.
   /* If node is a $PRIMARY\_NODE$ don't check it for Dominations */
   else if ($NODE = primary$)
   go to Step 5

5. If the SN node has not been checked with all the other nodes in $NODES$, go to Step 3, else go to Step 6.

6. If all the nodes in $SORT\_LIST$ have been checked go to Step 7, else go to Step 2.

7. Check if the nodes which are marked *primary* form a complete graph, if yes then return TRUE, else return FALSE. Here only the nodes which are marked *primary* need to be checked, because once a domination is found

the dominated node is marked as *secondary*, and when a node has been marked *secondary*, it is never checked to see if this node dominates some other node, it is only checked to see if this node is dominated by some other node. But this is the same as removing a dominated node from the graph, hence only check the nodes which are marked as *primary* to see if they form a complete graph.

6.4.3.2 Pseudo Code for Function *check_dominations*

```
check_dominations()
{
    int     node_id, dominated;
    int     sort_list;    /* Array containing the Sorted nodes */
    NODE    graph;    /* Array of pointers to Graph Nodes */
                  /* A Graph Node was shown in Figure 6.7 */
   /* Sort the nodes and save the order in array sort_list */
    sort_list = sort_nodes();
    /* Loop for the number of nodes */
    for ( i = 1; i ≤ number_of_nodes; i++ )
    {
        /* Take a node from the sort_list */
        node_id = sort_list[i]
        /* If this node is a primary node do not continue */
       if (graph[node_id]→primary ≠ 1 )
       {
            /* If node not already colored */
            if ( graph[node_id]→status ≠ 1 )
            {
                /* Since this node has been selected mark it as a primary node */
                graph[node_id]→prim = 1;
                /* Loop for this node with all the nodes */
```

```
            for ( dominated = 0; dominated ≤ number_of_nodes ;
dominated++ )
            {
                if ( domination_found(node_id, dominated) )
                {
                    /* Store the information of dominating and dominated
nodes */
                    mark_dominating_node( node_id, dominated );
                   /* Mark the dominated node as being a secondary node */
                    graph[dominated]→secondary = 1;
                }
            }
        }
      }
    }
     /* Now check if the nodes marked primary form a complete graph */
     if ( complete_graph() ) /* Function checks if graph is complete */
          return TRUE;
     else
          return FALSE;
}
```

6.4.4 Some Features of *MISDOM*

MISDOM is a fast program which uses the principle of dominations to color a graph. Since this method uses the same principle of dominations explained earlier, *MISDOM* will not only give us a Multi-Coloring of a Graph, but will also find the exact minimum solution for all non-cyclic graphs. The algorithm for *MISDOM* is very similar to the algorithm for *DOM* presented earlier in Chapter 3. The way *MISDOM* handles cyclic graphs is exactly the same as the way *DOM* handles cyclic graphs.

Theorem 6.3 *If the incompatibility graph being colored is a non-cyclic graph then the independent sets produced by* MISDOM *will all be the Maximum Independent Sets. This means that each node in the incompatibility graph will be colored by the maximum number of colors that it can be colored with.*

Proof 6.3 *In order to check all the possible colors that a node can be colored with, the node must be checked with all the other nodes in the graph. But this is exactly what the function* check_dominations *does. When function* check_dominations *checks for dominations, it starts with the node with the most number of edges and then checks if it dominates any other nodes in the graph. Now this dominating node is not checked to see if any other node dominates it, but no node can possibly dominate this node because this node has the most number of edges. Also since the function* check_dominations *does not remove a dominated node from the graph once a domination is found, that dominated node is checked with all the other nodes too. Hence if the graph is a non-cyclic graph, then Theorem 6.3 will always be true.*

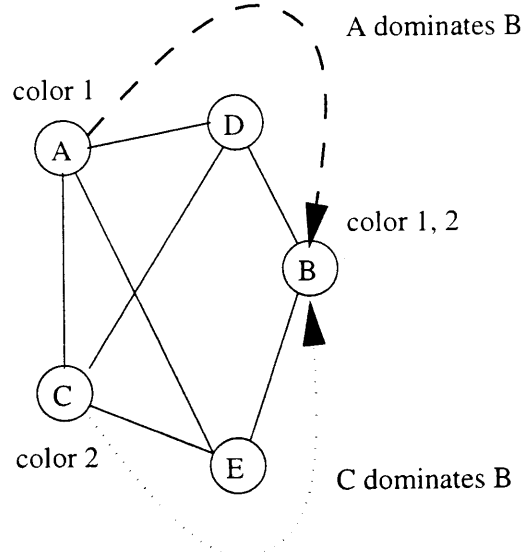


Figure 6.8: Reason why a dominated node can be colored all the colors of the nodes which dominate it

Theorem 6.4 MISDOM *will never produce Nodes with conflicting Colors when it colors a graph.*

Proof 6.4 *Consider the nodes "A" and "C" shown in Figure 6.8 which are incompatible with each other. As can be seen from Figure 6.8 node "A" dominates node "B", which means that node "B" does not have an edge with node "A". Since node "C" also dominates node "B" it means that node "B" does not have an edge with node "C". Thus node "B" can be colored the same colors as the colors assigned to nodes "A" and node "C". Thus proving that there will never be a conflict.*

The primary difference between *MISDOM* and *DOM* is that once a domination is found then the dominated node is not removed from the incompatibility graph. This is because only if it is found that these dominated nodes are dominated by some other node, can a Multi-Coloring of the incompatibility graph be found.

6.5 Summary and Conclusions of Chapter 6

In this Chapter the concept of Relations, and decomposition of relations was presented, and it was also shown how a Multi-Coloring can improve the quality of decompositions achieved. As the next step we actually have to test the Multi-Coloring in order to see if it really produces relations, and of what kind, in the decomposition of benchmark functions. This is presented in the next Chapter.

CHAPTER 7

AN EVALUATION OF THE MULTI-COLORING PROGRAM

In this Chapter the testing of the Multi-Coloring is done in order to verify if the Multi-Coloring can produce relations from functions and thus improve the quality of the decompositions achieved This strategy has been taken among other reasons because we have no examples of real-life relations, and we did not want to test on randomly generated data only. In this Chapter first the Multi-Coloring is tested on random graphs and then the Multi-Coloring is tested on graphs generated from real life functions. Testing is done on both MCNC, and Machine Learning benchmarks. When testing the Multi-Coloring in decomposition it was realized that a new method of evaluating the cost function was required. Hence in this Chapter a new approach to calculate the cost function of a decomposed function or relation is presented.

7.1 An Evaluation of Running *MISDOM* on randomly generated graphs

In this section *MISDOM* is tested on randomly generated graphs. These graphs were generated with nodes from 10 to 100 and with edge percent from 10% to 90%.

Table 7.1 shows how *MISDOM* performs on randomly generated graphs. In Table 7.1 the following notations are used:

C stands for the number of *colors* found.

NMOS stands for Number of *nodes in more than one set.* When *MISDOM* is used to calculate the column multiplicity in decomposition, **NMOS** will indicate the number of minterms in the G block which correspond to generalized don't cares.

Nodes vs Edges												
Edge%		*Nodes*										AVPCEP
		10	20	30	40	50	60	70	80	90	100	
10%	C	2	3	4	4	5	6	7	6	7	8	
	NMOS	5	11	15	14	20	32	31	29	40	46	44.2
	MCON	2	3	5	5	4	4	5	4	4	4	
20%	C	2	5	5	7	7	8	8	10	9	11	
	NMOS	0	9	11	20	17	21	27	31	24	35	35.5
	MCON	0	5	4	10	4	4	4	4	4	5	
30%	C	3	5	6	7	9	10	11	11	14	14	
	NMOS	3	9	7	11	17	22	21	14	30	29	29.6
	MCON	2	4	3	3	4	3	4	4	5	5	
40%	C	3	6	9	8	10	13	15	16	16	18	
	NMOS	1	7	12	8	12	19	22	27	22	28	28.7
	MCON	2	3	4	5	3	3	5	5	4	4	
50%	C	4	7	9	11	12	16	16	20	21	21	
	NMOS	3	7	8	10	13	19	18	25	29	26	28.7
	MCON	2	3	4	3	4	4	5	5	5	3	
60%	C	5	9	11	13	15	17	20	21	22	26	
	NMOS	2	6	8	8	13	14	15	21	16	26	23.45
	MCON	3	4	3	3	3	3	4	3	3	4	
70%	C	6	9	13	16	19	18	23	28	28	31	
	NMOS	1	6	6	9	12	10	19	20	18	20	22
	MCON	3	2	3	5	4	3	3	4	4	4	
80%	C	6	9	15	17	21	24	25	31	34	37	
	NMOS	1	4	7	5	8	12	8	16	15	16	16.7
	MCON	2	2	3	3	4	3	4	4	4	4	
90%	C	7	11	16	21	27	31	32	38	41	48	
	NMOS	0	1	2	5	9	12	7	12	9	17	13
	MCON	0	2	2	2	3	3	2	3	4	4	

Table 7.1: An Evaluation of running *MISDOM* on Randomly generated graphs

MCON stands for the *maximum colors assigned to one node.* When *MISDOM* is used to calculate the column multiplicity in Functional Decomposition, **MCON** will indicate the input cube or input minterm which has the largest number of values in the output.

Two trends can be observed from this Table. On moving along a row of the Table 7.1, that is for increasing nodes for a constant edge percent, *NMOS* increases,

and on moving along a column of the Table 7.1, that is for increasing edge percent for a constant number of nodes, **NMOS** decreases. In the last column of Table 7.1 **AVPCEP** stands for the *average percent for constant edge percent.* This is obtained by totaling all the **NMOS** in one row and dividing it by the total of the number of nodes. Hence this indicates what is the average percent of **NMOS** for a given percent of edges in the graph. From this Table it can be seen that *MISDOM* generates Maximum Independent sets (explained in Chapter 6 Section 6.1.2) with a large number of overlapping minterms. This is very desirable from our perspective because we want to create more generalized don't cares in the G functions. How generalized don't cares can improve the quality of achieved decompositions was introduced in Chapter 6.
As the next step it was required to test *MISDOM* on graphs generated during the process of Functional Decomposition to see if it can generate large overlaps in the Maximum Independent Sets in the graphs that are generated during Functional Decomposition, and thus produce a higher number of generalized don't cares.

7.2 A New Approach to calculate the cost function of a Decomposed Function or Relation

In Chapter 5 *DFC* was introduced for binary functions and for multi-valued functions. The *DFC* cost function has been proved to be a good evaluation of the cost function for binary functions in Machine Learning, but *DFC* does not provide a good evaluation for multi-valued functions and for relations. Also *DFC* does not take into account the circuit complexity of the decomposed function. By circuit complexity we mean how easy or how difficult it would be to realize this circuit in terms of gates. It was decided therefore to develop a new cost function which would not only consider the generalized don't cares while evaluating the cost, but would also take into account the circuit complexity of the decomposed function. Hence there are two factors being considered in the cost function:

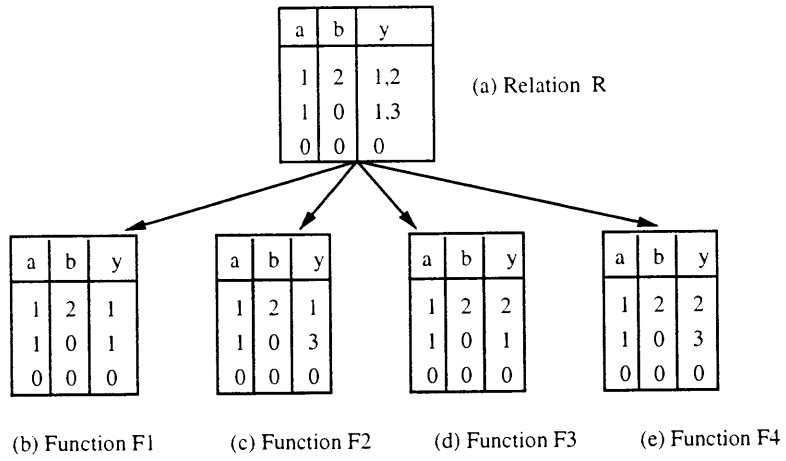


Figure 7.1: Different classes of Functions possible from a Relation

1. The first factor calculates the total number of functions obtained by selecting different don't cares of the relation. This cost function is called *NOFP*, which stands for *number of functions possible.*

2. The second factor taken into account is the simplicity of the Karnaugh map of the function or relation. This cost function has been given the name *COSC*, which stands for *cost of simplest circuit.*

The Cost Functions *NOFP*, and *COSC* are explained in the next two Sections with the help of examples.

7.2.1 Calculating *NOFP* for a Relation and a Function

NOFP takes into account the number of combinations obtained by the introduction of generalized don't cares in a function. If, for example, a generalized don't care in a **relation** has two possible values, it means that there are now two possible **functions**, one for each of the values of the generalized don't cares. Hence *NOFP* calculates the number of all possible combinations of classes of **functions** that can be obtained from a **relation**, by making choices in the generalized don't cares in the **relation**. Each class of function obtained from a

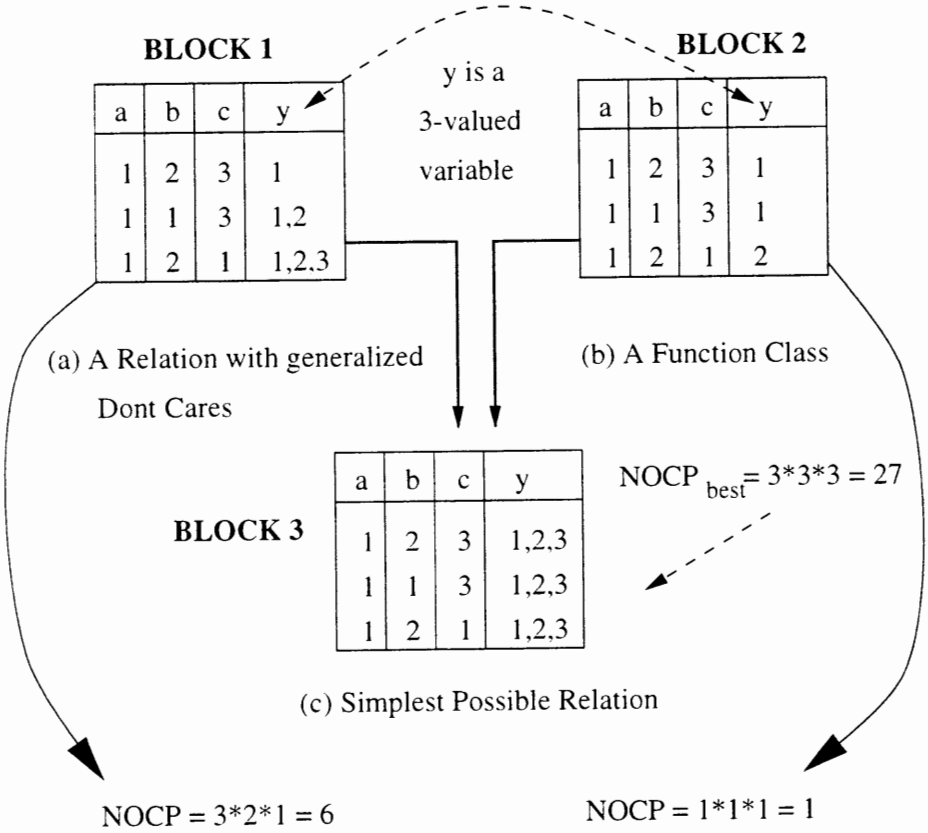


Figure 7.2: A Comparison of the calculation of *NOFP* for a Relation and for a Function

Relation will have the same inputs but different outputs. Figure 7.1(a) shows a relation with two inputs and one output. Figure 7.1(b), 7.1(c), 7.1(d) and 7.1(e) show the four possible functions obtained from the relation in Figure 7.1(a). As can be seen each of these functions is a subset of the original relation. Figure 7.2 shows how *NOFP* is calculated. Figure 7.2(a) shows a relation having three three-valued inputs, and one three-valued output. Figure 7.2(b) shows a function which has three three-valued inputs and one three-valued output. As can be seen, the function and the relation both have the same inputs, the function has only one possible output, while the relation has generalized don't cares in the output.

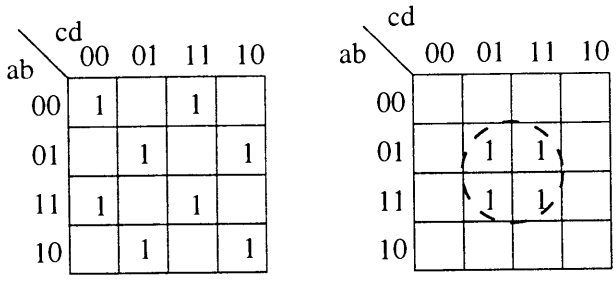
Definition 7.1 *For a block with number of multi-values in the output =* **nmv**, *and number of rows in the block =* **rows**, *the maximum number of combinations of maps possible for this block ($NOFP_{best}$) is defined as* $NOFP_{best} = (nmv)^{rows}$

The block labeled BLOCK 3 in Figure 7.2(c) shows $NOFP_{best}$ for the blocks labeled BLOCK 1 and BLOCK 2 in Figure 7.2(a) and Figure 7.2(b), respectively. Now for BLOCK 1 the number of possible combinations is 6, hence *NOFP* for the relation shown in Figure 7.2(a) is 6, and *NOFP* for the function shown in Figure 7.2(b) is 1. For a function *NOFP* will always be equal to one. The higher the value of *NOFP*, the better the block.

7.2.2 Calculating *COSC* for a Relation and a Function

COSC calculates what is the cost of implementing a decomposed function or relation. *COSC* considers two factors in finding out the cost of the function or relation:

The first factor considered is the number of overlaps in the output, where two minterms overlap if they have a common output. But this does not really illustrate how simple it would be to realize the function or relation. Figure 7.3(a) shows a function with 8 minterms having the same output one, hence it has 8 overlaps, and Figure 7.3(b) shows a function having 4 minterms having the same output one, hence it has 4 overlaps. Now, as can be seen from Figure 7.3, even



(a) Function with 8 repetitions.

(b) Function with 4 repetitions

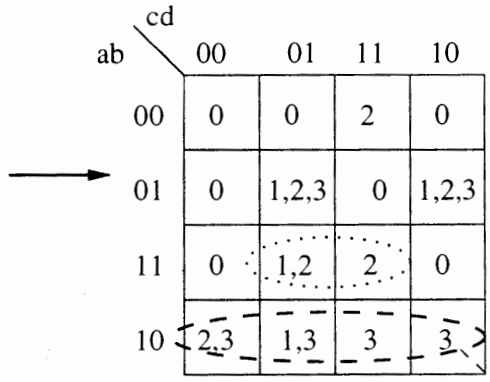
Figure 7.3: A Comparison showing that more repetions of common variables in not necessarily better

though the function in Figure 7.3(a) has more overlaps than the function in Figure 7.3(b), the function in Figure 7.3(b) would be simpler to realize in terms of gates than the function shown in Figure 7.3(a).

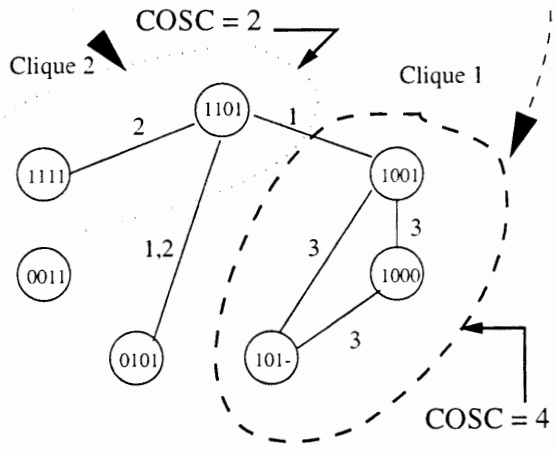
In Figure 7.3 it was shown that more overlaps is not necessarily a good thing. What is needed, is to check if for every set of two cubes that overlap the Hamming distance of these two cubes is equal to one? This is the second factor considered while calculating *COSC*. Figure 7.4 shows how *COSC* can be calculated for a relation. Figure 7.4(a) shows a Table of a relation F having three inputs and one output. Figure 7.4(b) shows the Karnaugh map of F. Now in order to calculate *COSC* for F, a compatibility graph is created. In the compatibility graph each node represents a row of the Table of the relation F. In this compatibility graph any two nodes $N1$ and $N2$ have an edge between them if nodes $N1$ and $N2$ have a Hamming distance of one, and if they have a common output. Thus since minterm "1101" and minterm "1111" have a Hamming distance of one, and since they both have a common output which is a "2", they have an edge. The edges weight shown in the compatibility graph correspond to the output(or outputs) common to the nodes having this edge. In order to calculate *COSC*, each minterm in the relation is checked with all the other minterms in the relation and if they can have an edge then the cost *COSC* is incremented by two(because two minterms

a	b	c	d	y
1	1	0	1	1,2
1	1	1	1	2
1	0	0	1	1,3
0	1	0	1	1,2,3
1	0	0	0	2,3
1	0	1	-	3
0	0	1	1	2
0	0	0	-	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	1	0	0	0
1	1	1	0	0

(a) Table of Relation F



(b) Karnaugh Map of Relation F



(c) Compatibility graph of Relation F

Figure 7.4: Example showing how *COSC* is calculated

are being considered). Thus *COSC* reflects the number of minterms which can be combined with other minterms. For Clique 1 shown in Figure 7.4(c) the cost is two since two minterms are being combined. For Clique 2 shown in Figure 7.4(c) the cost is four since there are four possible combinations of the minterms. In the calculation of *COSC* all the cliques are considered while calculating the cost, including the overlapping cliques.

7.3 An Evaluation of Running *MISDOM* on Graphs generated during Decomposition of Functions and Relations

In this section a comparison is done to see if *MISDOM* can improve the quality of decompositions achieved. The cost factors *NOFP* and *COSC* will be used to evaluate the quality of the decomposed functions. A comparison is done on functions from Circuit, and Machine Learning Benchmarks. In order to do the comparison it was needed to compare a Graph Coloring and a Multi-Coloring. For this experiment *DOM* was chosen as the Graph Coloring program, and the Multi-Coloring program is *MISDOM*. The results of the testing has been divided into two parts, first the testing is done on the Circuit benchmarks, and then the testing is done on the Machine Learning Benchmarks. *MVGUD* was tested with two, four, and five variables in the Bound set. The following points help to understand the values of *NOFP* and *COSC* in all the Tables shown.

1. While analyzing these Tables, it should be remembered, that the higher the values of *NOFP* and *COSC*, the better is the decomposition.

2. The values shown for *NOFP* and *COSC* in the Tables are the sums of the values of *NOFP* and *COSC* for each block in the decomposed functions or relations.

3. If the value of *NOFP* is equal to one, it means that there were no generalized don't cares in the output for the decomposed function or relation. When

Function				Algorithm			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
5xp1	7	10	143	1	250	2	224
9sym1	9	1	158	1	62	1	1372
b12	15	9	172	1	280	1	248
con1	7	2	18	1	96	1	70
bw	5	28	97	1	618	25	718
ex5p	8	63	214	1	2488	2	3272
misex1	8	7	40	1	214	1	334
rd53	5	3	63	1	46	1	48
rd73	7	3	274	1	52	1	78
rd84	8	4	515	1	106	1	170
sao2	10	4	133	1	512	1	526
squar5	5	8	56	1	166	1	172
xor5	5	1	32	1	1	1	1

Table 7.2: *MVGUD* run with 2 variables in the bound set on MCNC Benchmarks

DOM is used in *MVGUD* to find the column multiplicity, *NOFP* will always be equal to one.

4. If the value of *COSC* is equal to one it means that there were no cubes or minterms in the decomposed blocks of the function that could be combined. Thus if the value of *COSC* is one for a function it implies that this function would be very costly in terms of gates to implement.

7.3.1 A Comparison of *MISDOM* and *DOM* on MCNC Benchmarks

Table 7.2 shows the results of running *MVGUD* with two variables in the bound set, Table 7.3 shows the results of running *MVGUD* with four variables in the bound set, and Table 7.4 shows the results of running *MVGUD* with five variables in the bound set. Looking at the values of *NOFP* in Table 7.2 it can be seen that *MISDOM* finds a *NOFP* better than one for only three benchmarks. For a total number of program runs of fourteen, *DOM* has a better *COSC* in three cases, while *MISDOM* is better in eleven cases. *DOM* and *MISDOM* will generate

Function				Algorithm			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
5xp1	7	10	143	1	914	1	532
9sym1	9	1	158	1	188	1	572
b12	15	9	172	1	784	1	658
con1	7	2	18	1	80	1	80
bw	5	28	97	1	1162	14	1314
ex5p	8	63	214	1	7228	1	8782
misex1	8	7	40	1	794	1	940
rd53	5	3	63	1	60	1	70
rd73	7	3	274	1	616	1	596
rd84	8	4	515	1	214	1	360
sao2	10	4	133	1	4726	1	2364
squar5	5	8	56	1	354	1	360
xor5	5	1	32	1	1	1	1

Table 7.3: *MVGUD* run with 4 variables in the bound set on MCNC Benchmarks

Function				Algorithm			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
5xp1	7	10	143	1	822	1	804
9sym1	9	1	158	1	1842	1	2422
b12	15	9	172	1	1574	1	1502
con1	7	2	18	1	200	1	200
bw	5	28	97	1	2236	1	2236
ex5p	8	63	214	1	19832	1	21010
misex1	8	7	40	1	896	1	864
rd53	5	3	63	1	1850	1	1850
rd73	7	3	274	1	190	1	228
rd84	8	4	515	1	928	1	1120
sao2	10	4	133	1	3016	1	3248
squar5	5	8	56	1	782	1	782
xor5	5	1	32	1	1	1	1

Table 7.4: *MVGUD* run with 5 variables in the bound set on MCNC Benchmarks

the same number of colors for the graphs, but due to the different strategies employed in the two programs, for the same incompatibility graph *DOM* might give the nodes different colors than *MISDOM*, so the columns of the Karnaugh map of the function, corresponding to the nodes of the incompatibility graph will have different codes assigned to them. Hence due to this the Karnaugh map of the decomposed function can be different for when *DOM* is used or when *MISDOM* is used. Thus it is possible for *DOM* to have a better *COSC* than *MISDOM*, but in most cases *COSC* for *MISDOM* will be better.

Looking at the values of *NOFP* for *MISDOM* in Tables 7.3 and 7.3, it can be seen that *MISDOM* fails to introduce generalized don't cares in most cases. The reason why *MISDOM* fails to introduce generalized don't cares in most cases for *MCNC* benchmarks is because these benchmarks have very few don't cares, and the graphs generated during decomposition for these benchmarks are dense graphs, hence in most cases the nodes have only one possible color.

7.3.1.1 Summary of the results of testing *MISDOM* on MCNC benchmarks

The primary advantage, that we want to gain by using a Multi-Coloring in Decomposition is the introduction of generalized don't cares. Hence from the results of testing on the *MCNC* benchmarks it can be concluded that the Multi-Coloring will not improve the quality of decompositions achieved on circuit benchmarks. Now it was required to see if the results differ on Machine Learning Benchmarks. The results of testing on Machine Learning Benchmarks is presented in the next section.

7.3.2 A Comparison of *MISDOM* and *DOM* on Machine Learning Benchmarks

Two experiments were performed here: In Experiment 1 the number of variables in the Bound Set were increased and *DOM* and *MISDOM* were compared, in this experiment the number of don't cares in the Machine Learning Functions were kept constant. In Experiment 2 the number of don't cares in the functions were increased and then *DOM* and *MISDOM* were compared by running *MVGUD* with different variables in the Bound Set.

7.3.2.1 Experiment 1: Running *MVGUD* on 8 variable Machine Learning Benchmarks with 70% don't cares

In this Experiment *MVGUD* was run with *MISDOM* and *DOM* on the 8 variable Machine Learning Benchmarks, and the number of don't cares in these Benchmarks was 70% and the number of variables in the Bound Set was increased. Tables 7.5, 7.6, 7.7, and 7.8 show the results of running *MVGUD* with two, four, five, and six variables in the Bound Set respectively. Here a big difference is observed when compared to the results obtained on MCNC benchmarks. As can be seen from these Tables, *MISDOM* is able to find overlaps in the graphs generated during decomposition and is thus able to introduce generalized don't cares in the decomposed functions. Also it is observed from these Tables that as the size of the Bound Set increases the number of generalized don't cares introduced also increases. Also from these Tables it is observed that whenever *NOFP* is high, the *COSC* for the benchmark is also very good, which is obvious because now there are more possible combinations, and thus the decomposed functions can be much simpler.

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				Algorithm			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	36	1	46
add2	8	1	77	1	28	2	42
add4	8	1	77	1	4	1	4
and_or_chain8	8	1	77	1	20	1	28
ch15f0	8	1	77	1	52	32	100
ch176f0	8	1	77	1	16	1	16
ch177f0	8	1	77	1	42	1	40
ch22f0	8	1	77	1	44	10	74
ch30f0	8	1	77	1	38	4	50
ch47f0	8	1	77	1	48	219	78
ch52f4	8	1	77	1	24	2	34
ch70f3	8	1	77	1	34	4	74
ch74f1	8	1	77	1	48	22	98
ch83f2	8	1	77	1	20	4	32
ch8f0	8	1	77	1	44	10	82
contains_4_ones	8	1	77	1	36	6	40
greater_than	8	1	77	1	76	19	226
interval1	8	1	77	1	60	10	80
interval2	8	1	77	1	12	1	28
kdd1	8	1	77	1	20	1	28
kdd2	8	1	77	1	12	1	12
kdd3	8	1	77	1	1	1	1
kdd4	8	1	77	1	34	2	64
kdd5	8	1	77	1	12	1	12
kdd6	8	1	77	1	44	2	88
kdd7	8	1	77	1	8	1	8
kdd8	8	1	77	1	20	1	40
kdd9	8	1	77	1	38	1	38
majority_gate	8	1	77	1	48	6	60
modulus2	8	1	77	1	44	6	56
monkish1	8	1	77	1	8	1	8
monkish2	8	1	77	1	18	1	18
monkish3	8	1	77	1	22	2	48
mux8	8	1	77	1	30	2	40
nnr1	8	1	77	1	46	8	68
nnr2	8	1	77	1	14	1	14
nnr3	8	1	77	1	68	276	110
or_and_chain8	8	1	77	1	20	1	30
pal	8	1	77	1	28	1	46
pal_dbl_output	8	1	77	1	62	78	138
parity	8	1	77	1	1	1	1
primes8	8	1	77	1	1	1	1
remainder2	8	1	77	1	86	10	78
rnd1	8	1	77	1	100	4	126
rnd2	8	1	77	1	52	160	160
rnd3	8	1	77	1	56	48	128

Table 7.5: *MVGUD* run with 2 variables in the bound set on 8 variable Machine Learning Benchmarks with 70% don't cares

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				Algorithm			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	1	4	102
add2	8	1	77	1	82	280	140
add4	8	1	77	1	32	1	32
and_or_chain8	8	1	77	1	64	1	108
ch15f0	8	1	77	1	66	80	138
ch176f0	8	1	77	1	84	1	88
ch177f0	8	1	77	1	32	1	106
ch22f0	8	1	77	1	82	1	82
ch30f0	8	1	77	1	76	24	134
ch47f0	8	1	77	1	72	66	112
ch52f4	8	1	77	1	66	8	112
ch70f3	8	1	77	1	102	64	126
ch83f2	8	1	77	1	60	40	96
ch8f0	8	1	77	1	88	1	100
contains_4_ones	8	1	77	1	40	34	70
greater_than	8	1	77	1	72	258	136
interval1	8	1	77	1	80	20	120
interval2	8	1	77	1	92	16	332
kdd1	8	1	77	1	18	1	18
kdd2	8	1	77	1	96	1	96
kdd3	8	1	77	1	78	1	100
kdd4	8	1	77	1	20	1	20
kdd5	8	1	77	1	88	128	146
kdd6	8	1	77	1	56	1	56
kdd7	8	1	77	1	80	1	86
kdd8	8	1	77	1	16	1	16
kdd9	8	1	77	1	76	1	94
majority_gate	8	1	77	1	62	68	116
modulus2	8	1	77	1	68	34	134
monkish1	8	1	77	1	32	1	32
monkish2	8	1	77	1	46	8	72
monkish3	8	1	77	1	80	1	106
mux8	8	1	77	1	92	136	148
nnr1	8	1	77	1	76	1	76
nnr2	8	1	77	1	40	1	40
nnr3	8	1	77	1	96	332	160
or_and_chain8	8	1	77	1	104	1	104
pal	8	1	77	1	80	1	102
pal_dbl_output	8	1	77	1	80	4114	162
parity	8	1	77	1	1	1	1
primes8	8	1	77	1	1	1	1
remainder2	8	1	77	1	60	304	140
rnd1	8	1	77	1	66	2064	160
rnd2	8	1	77	1	80	8204	146
rnd3	8	1	77	1	92	262176	210

Table 7.6: *MVGUD* run with 4 variables in the bound set on 8 variable Machine Learning Benchmarks with 70% don't cares

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				Algorithm			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	110	64	176
add2	8	1	77	1	88	1	88
add4	8	1	77	1	132	1	132
and_or_chain8	8	1	77	1	140	1	258
ch15f0	8	1	77	1	126	13824	352
ch176f0	8	1	77	1	128	1	210
ch177f0	8	1	77	1	126	1	138
ch22f0	8	1	77	1	134	1	150
ch30f0	8	1	77	1	128	1024	278
ch47f0	8	1	77	1	128	128	316
ch52f4	8	1	77	1	136	18432	610
ch70f3	8	1	77	1	112	8192	454
ch74f1	8	1	77	1	112	373248	436
ch83f2	8	1	77	1	118	32768	316
ch8f0	8	1	77	1	72	256	210
contains_4_ones	8	1	77	1	92	1	192
greater_than	8	1	77	1	116	1024	246
interval1	8	1	77	1	142	62208	390
interval2	8	1	77	1	14	1	14
kdd1	8	1	77	1	142	1	204
kdd2	8	1	77	1	146	1	146
kdd3	8	1	77	1	16	1	16
kdd4	8	1	77	1	142	16	332
kdd5	8	1	77	1	122	1	198
kdd6	8	1	77	1	142	16	212
kdd7	8	1	77	1	128	1	224
kdd8	8	1	77	1	134	64	304
kdd9	8	1	77	1	152	8192	326
majority_gate	8	1	77	1	94	256	342
modulus2	8	1	77	1	124	512	392
monkish1	8	1	77	1	132	1	132
monkish2	8	1	77	1	94	4096	236
monkish3	8	1	77	1	110	1	222
mux8	8	1	77	1	144	64	230
nnr1	8	1	77	1	132	64	238
nnr2	8	1	77	1	150	1	150
nnr3	8	1	77	1	156	1492992	644
or_and_chain8	8	1	77	1	134	1	232
pal	8	1	77	1	112	1	250
pal_dbl_output	8	1	77	1	118	124416	380
parity	8	1	77	1	1	1	1
primes8	8	1	77	1	1	1	1
remainder2	8	1	77	1	98	746496	318
rnd1	8	1	77	1	130	995328	460
rnd2	8	1	77	1	124	106168320	452
rnd3	8	1	77	1	146	165888	426

Table 7.7: *MVGUD* run with 5 variables in the bound set on 8 variable Machine Learning Benchmarks with 70% don't cares

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				**Algorithm**			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	174	1	174
add2	8	1	77	1	80	1	80
add4	8	1	77	1	168	1	168
and_or_chain8	8	1	77	1	202	1	202
ch15f0	8	1	77	1	126	1	126
ch176f0	8	1	77	1	162	2097152	232
ch177f0	8	1	77	1	162	1	162
ch22f0	8	1	77	1	118	67108864	240
ch30f0	8	1	77	1	120	32768	208
ch47f0	8	1	77	1	76	18874368	250
ch52f4	8	1	77	1	94	1729626112	322
ch74f1	8	1	77	1	144	256	218
ch83f2	8	1	77	1	104	2031390720	290
ch8f0	8	1	77	1	164	131072	206
contains_4_ones	8	1	77	1	60	131072	136
greater_than	8	1	77	1	144	8192	186
interval1	8	1	77	1	96	1820327936	282
interval2	8	1	77	1	124	1586446336	286
kdd1	8	1	77	1	170	1	170
kdd2	8	1	77	1	220	1	220
kdd3	8	1	77	1	164	1	164
kdd4	8	1	77	1	4	1	4
kdd5	8	1	77	1	186	1	186
kdd6	8	1	77	1	204	1	204
kdd7	8	1	77	1	146	32	164
kdd8	8	1	77	1	156	524288	228
kdd9	8	1	77	1	98	63700992	236
kdd10	8	1	77	1	132	1	132
majority_gate	8	1	77	1	108	1787822080	266
modulus2	8	1	77	1	112	1146617856	280
monkish1	8	1	77	1	172	1	172
monkish2	8	1	77	1	94	1820327936	280
monkish3	8	1	77	1	188	1	188

Table 7.8: *MVGUD* run with 6 variables in the bound set on 8 variable Machine Learning Benchmarks with 70% don't cares

7.3.2.2 Experiment 2: Running *MVGUD* on 8 variable Machine Learning Benchmarks with 90% don't cares

In this Experiment, the don't cares in the eight variable Machine Learning Benchmarks were increased to 90% and *MVGUD* was tested on these Benchmarks with two, four, five, and six variables in the Bound set.

Tables 7.9, 7.10, 7.11, and 7.12 show the results for testing *MVGUD* with two, four, five, and six variables in the Bound set respectively. On comparing these Tables with the Tables from Experiment 1 it is seen that as the number of don't cares increases, *NOFP* also increases.

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				**Algorithm**			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	24	2	26
add2	8	1	77	1	18	5	28
add4	8	1	77	1	1	1	1
and_or_chain8	8	1	77	1	12	1	12
ch15f0	8	1	77	1	44	14	82
ch176f0	8	1	77	1	16	1	16
ch177f0	8	1	77	1	1	1	1
ch22f0	8	1	77	1	14	1	14
ch30f0	8	1	77	1	12	1	12
ch47f0	8	1	77	1	18	4	26
ch52f4	8	1	77	1	28	4	42
ch74f1	8	1	77	1	16	2	18
ch83f2	8	1	77	1	26	10	66
ch8f0	8	1	77	1	1	1	1
contains_4_ones	8	1	77	1	14	6	44
greater_than	8	1	77	1	20	1	20
interval1	8	1	77	1	30	10	108
interval2	8	1	77	1	28	30	96
kdd1	8	1	77	1	16	4	32
kdd2	8	1	77	1	12	2	20
kdd3	8	1	77	1	16	1	16
kdd4	8	1	77	1	4	1	4
kdd5	8	1	77	1	20	1	20
kdd6	8	1	77	1	12	2	20
kdd7	8	1	77	1	16	2	24
kdd8	8	1	77	1	18	1	18
kdd9	8	1	77	1	14	4	22
kdd10	8	1	77	1	30	2	24
majority_gate	8	1	77	1	28	8	58
modulus2	8	1	77	1	18	1	18
monkish1	8	1	77	1	1	1	1
monkish2	8	1	77	1	20	2	38
monkish3	8	1	77	1	18	2	22

Table 7.9: *MVGUD* run with 2 variables in the bound set on 8 variable Machine Learning Benchmarks with 90% don't cares

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				**Algorithm**			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	44	24	62
add2	8	1	77	1	26	6	56
add4	8	1	77	1	46	8	56
and_or_chain8	8	1	77	1	34	1	34
ch15f0	8	1	77	1	42	72	76
ch176f0	8	1	77	1	36	8	62
ch177f0	8	1	77	1	60	48	82
ch22f0	8	1	77	1	28	1	28
ch30f0	8	1	77	1	58	132	90
ch47f0	8	1	77	1	38	32	66
ch52f4	8	1	77	1	46	24	82
ch74f1	8	1	77	1	48	256	80
ch83f2	8	1	77	1	54	36	92
ch8f0	8	1	77	1	1	1	1
contains_4_ones	8	1	77	1	28	136	82
greater_than	8	1	77	1	64	1	64
interval1	8	1	77	1	28	40	68
interval2	8	1	77	1	36	48	76
kdd1	8	1	77	1	26	1	26
kdd2	8	1	77	1	20	1	20
kdd3	8	1	77	1	56	8	76
kdd4	8	1	77	1	1	1	1
kdd5	8	1	77	1	56	34	82
kdd6	8	1	77	1	42	1	42
kdd7	8	1	77	1	30	34	56
kdd8	8	1	77	1	10	1	10
kdd9	8	1	77	1	40	24	72
kdd10	8	1	77	1	44	40	92
majority_gate	8	1	77	1	38	24	70
modulus2	8	1	77	1	52	18	70
monkish1	8	1	77	1	1	1	1
monkish2	8	1	77	1	38	36	70
monkish3	8	1	77	1	54	72	98

Table 7.10: *MVGUD* run with 4 variables in the bound set on 8 variable Machine Learning Benchmarks with 90% don't cares

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				**Algorithm**			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	56	64	100
add2	8	1	77	1	46	2	52
add4	8	1	77	1	50	4	62
and_or_chain8	8	1	77	1	36	8	82
ch15f0	8	1	77	1	32	2	44
ch176f0	8	1	77	1	42	1	42
ch177f0	8	1	77	1	56	16	70
ch22f0	8	1	77	1	12	1	12
ch30f0	8	1	77	1	46	64	90
ch47f0	8	1	77	1	34	8	72
ch52f4	8	1	77	1	64	64	90
ch74f1	8	1	77	1	56	4096	70
ch83f2	8	1	77	1	52	32	80
ch8f0	8	1	77	1	1	1	1
contains_4_ones	8	1	77	1	42	4	54
greater_than	8	1	77	1	54	1	64
interval1	8	1	77	1	44	256	98
interval2	8	1	77	1	36	4	72
kdd1	8	1	77	1	14	1	14
kdd2	8	1	77	1	18	1	18
kdd3	8	1	77	1	52	16	78
kdd4	8	1	77	1	16	1	16
kdd5	8	1	77	1	52	2	76
kdd6	8	1	77	1	56	2048	94
kdd7	8	1	77	1	42	8	66
kdd8	8	1	77	1	52	4	82
kdd9	8	1	77	1	40	16	70
kdd10	8	1	77	1	56	4	68
majority_gate	8	1	77	1	38	4	76
modulus2	8	1	77	1	42	8	62
monkish1	8	1	77	1	1	1	1
monkish2	8	1	77	1	60	512	88
monkish3	8	1	77	1	58	4096	110

Table 7.11: *MVGUD* run with 5 variables in the bound set on 8 variable Machine Learning Benchmarks with 90% don't cares

Comparison of Multi-Coloring and Coloring on Machine Learning Benchmarks							
Function				**Algorithm**			
				DOM		*MISDOM*	
name	in	out	cubes	*NOFP*	*COSC*	*NOFP*	*COSC*
add0	8	1	77	1	56	512	60
add2	8	1	77	1	42	1	42
add4	8	1	77	1	50	1	50
and_or_chain8	8	1	77	1	44	1024	70
ch15f0	8	1	77	1	30	4	34
ch176f0	8	1	77	1	44	1	44
ch177f0	8	1	77	1	42	1	42
ch22f0	8	1	77	1	38	64	60
ch30f0	8	1	77	1	52	64	54
ch47f0	8	1	77	1	26	32	44
ch52f4	8	1	77	1	48	256	62
ch74f1	8	1	77	1	50	256	74
ch83f2	8	1	77	1	42	1	40
ch8f0	8	1	77	1	1	1	1
contains_4_ones	8	1	77	1	28	8	38
greater_than	8	1	77	1	56	1	56
interval1	8	1	77	1	24	8	32
interval2	8	1	77	1	46	16	60
kdd1	8	1	77	1	30	256	54
kdd2	8	1	77	1	44	2048	56
kdd3	8	1	77	1	36	16	52
kdd4	8	1	77	1	4	1	4
kdd5	8	1	77	1	40	8	68
kdd6	8	1	77	1	36	16384	64
kdd7	8	1	77	1	64	1	64
kdd8	8	1	77	1	50	8	64
kdd9	8	1	77	1	32	62208	90
kdd10	8	1	77	1	42	1	42
majority_gate	8	1	77	1	32	128	66
modulus2	8	1	77	1	30	8	46
monkish1	8	1	77	1	1	1	1
monkish2	8	1	77	1	38	1	38
monkish3	8	1	77	1	28	512	56

Table 7.12: *MVGUD* run with 6 variables in the bound set on 8 variable Machine Learning Benchmarks with 90% don't cares

7.3.3 A Summary of the results of testing *MISDOM* on Machine Learning Benchmarks

The results obtained from the testing of *MISDOM* on Machine Learning Benchmarks have proved that Multi-Colorings or Clique-Coverings will nearly always produce better decompositions for Machine Learning Functions. Two example decomposed functions are shown here to show how the G functions and the H functions differ depending on whether a Multi-Coloring is used or a Graph Coloring is used.

Table 7.13 shows the G function and Table 7.14 shows the H function which were obtained by running *MVGUD* on the Machine Learning benchmark *psu_ch47f0* with five variables in the bound set. Tables 7.15 and 7.16 show the resulting G and H Tables respectively, for *MVGUD* run on the Machine Learning Function *psu_rnd3* with five variables in the bound set. These Tables show the actual output file from *MVGUD*. In these Tables the first column represents the decomposed function for *MVGUD* run with *MISDOM*, and the second column represents the decomposed function for *MVGUD* run with *DOM*. The output files are in the *mvblif* format. In this format:

.type: Stands for the type of file.

.i: Stands for the number of inputs.

.o: Stands for the number of outputs.

.ilb: Stands for the names of the inputs.

.ob: Stands for the names of the outputs.

.imv: Represents the maximum multi-value of each input.

.omv: Represents the maximum multi-value of each output.

.p: Stands for the number of cubes in the block. From the G Tables 7.13 and 7.15 the generalized don't cares can be seen in the output for when *MVGUD* is run with *MISDOM*.

MVGUD run with *MISDOM*						*MVGUD* run with *DOM*					
.type mv						.type mv					
.i 5						.i 5					
.o 1						.o 1					
.ilb a0 a1 a2 a4 a5						.ilb a0 a1 a2 a4 a5					
.ob s2.0						.ob s2.0					
.imv 2 2 2 2 2						.imv 2 2 2 2 2					
.omv 4						.omv 4					
.p 26						.p 26					
0	1	0	1	0	1	0	1	0	1	0	0
0	1	0	1	1	3	0	1	0	1	1	3
0	1	0	0	0	3	0	1	0	0	0	3
0	1	1	1	0	2	0	1	1	1	0	1
0	1	1	1	1	0,3	0	1	1	1	1	2
0	1	1	0	0	3	0	1	1	0	0	3
0	1	1	0	1	0,3	0	1	1	0	1	2
0	0	0	1	0	-	0	0	0	1	0	0
0	0	0	1	1	2	0	0	0	1	1	1
0	0	0	0	0	3	0	0	0	0	0	3
0	0	0	0	1	2,3	0	0	0	0	1	1
0	0	1	1	0	0	0	0	1	1	0	2
0	0	1	1	1	1,2	0	0	1	1	1	0
0	0	1	0	0	3	0	0	1	0	0	3
0	0	1	0	1	3	0	0	1	0	1	3
1	1	0	1	0	3	1	1	0	1	0	3
1	1	0	0	1	3	1	1	0	0	1	3
1	1	1	1	0	3	1	1	1	1	0	3
1	1	1	1	1	0,3	1	1	1	1	1	2
1	0	0	1	1	3	1	0	0	1	1	3
1	0	0	0	0	2,3	1	0	0	0	0	1
1	0	0	0	1	1	1	0	0	0	1	0
1	0	1	1	0	3	1	0	1	1	0	3
1	0	1	1	1	3	1	0	1	1	1	3
1	0	1	0	0	0	1	0	1	0	0	2
1	0	1	0	1	1,2	1	0	1	0	1	0
.end						.end					

Table 7.13: Resulting G Function for *MVGUD* run on Benchmark *psu_ch47f0*

.type mv .i 4 .o 1 .ilb a3 a6 a7 s2.0 .ob o0 .imv 2 2 2 4 .omv 2 .p 36					.type mv .i 4 .o 1 .ilb a3 a6 a7 s2.0 .ob o0 .imv 2 2 2 4 .omv 2 .p 36				
1	0	0	0,3	0	0	1	0	2	0
0	0	0	2	1	0	0	0	1	1
0	1	1	1	1	0	0	0	0	1
1	0	1	0,3	0	0	0	1	2	0
0	1	0	1,2	0	1	0	1	2	0
0	1	0	0,3	0	1	1	0	1	0
0	1	0	2	0	1	0	0	2	0
0	0	1	0,3	0	0	1	0	0	0
0	1	1	2,3	0	0	1	1	0	1
1	0	1	3	0	1	1	1	3	0
1	0	1	0	0	1	0	0	1	1
1	1	0	2,3	0	1	0	1	3	0
1	1	1	-	0	1	0	1	1	0
0	1	1	0	1	1	0	0	3	0
1	1	1	2,3	0	0	0	0	2	0
0	0	0	1	1	0	1	0	1	0
1	0	0	2	1	0	1	1	2	1
1	1	1	3	0	0	1	1	1	0
1	0	1	2,3	0	1	1	1	0	0
1	0	0	3	0	0	1	1	3	0
0	0	0	0	0	1	1	0	3	0
0	1	1	2	0	1	0	1	0	1
0	0	0	0,3	0	1	0	0	0	1
0	0	1	1	1	0	0	1	0	1
1	1	1	1,2	0	0	0	1	1	1
0	1	1	3	0	0	1	0	3	0
0	0	1	1,2	1	1	1	1	1	0
1	0	1	1	1	0	0	1	3	0
1	0	0	1,2	1	0	0	0	3	0
1	1	0	3	0	1	1	0	2	1
0	0	1	2	1	.end				
0	1	0	3	0					
1	1	1	2	0					
0	0	1	3	0					
0	0	0	3	0					
1	1	0	0	1					
.end									

Table 7.14: Resulting H Function for *MVGUD* run on Benchmark *psu_ch47f0*

MVGUD run with *MISDOM*						*MVGUD* run with *DOM*					
.type mv						.type mv					
.i 5						.i 5					
.o 1						.o 1					
.ilb a0 a2 a3 a4 a7						.ilb a0 a2 a3 a4 a7					
.ob s2.2						.ob s2.2					
.imv 2 2 2 2 2						.imv 2 2 2 2 2					
.omv 5						.omv 5					
.p 30						.p 31					
0	1	0	1	0	4	0	1	0	1	0	1
0	1	0	1	1	3	0	1	0	1	1	0
0	1	0	0	0	4	0	1	0	0	0	1
0	1	0	0	1	4	0	1	0	0	1	1
0	1	1	1	0	2	0	1	1	1	0	4
0	1	1	1	1	3	0	1	1	1	1	0
0	1	1	0	0	3	0	1	1	0	0	0
0	1	1	0	1	0,2,4	0	1	1	0	1	3
0	0	0	1	0	3	0	0	0	1	0	2
0	0	0	1	1	3	0	0	0	1	1	2
0	0	0	0	0	0,2,4	0	0	0	0	0	0
0	0	0	0	1	2	0	0	0	0	1	2
0	0	1	1	0	0,1,2	0	0	1	1	0	1
0	0	1	1	1	0,1,2	0	0	1	1	1	3
0	0	1	0	0	0,1	0	0	1	0	0	0
0	0	1	0	1	1,3	0	0	1	0	1	3
1	1	0	1	0	0	1	1	0	1	0	1
1	1	0	1	1	0	1	1	0	1	1	1
1	1	0	0	0	0,2,4	1	1	0	0	0	2
1	1	0	0	1	3,4	1	1	0	0	1	2
1	1	1	1	0	4	1	1	1	1	1	0
1	1	1	1	1	1,2	1	1	1	0	0	4
1	1	1	0	0	3	1	1	1	0	1	1
1	1	1	0	1	0,4	1	0	0	1	0	1
1	0	0	1	1	1	1	0	0	1	1	1
1	0	0	0	0	1	1	0	0	0	0	1
1	0	0	0	1	0	1	0	0	0	1	3
1	0	1	1	0	1,3	1	0	1	1	0	3
1	0	1	0	0	3,4	1	0	1	1	1	3
1	0	1	0	1	0,2	1	0	1	0	0	0
.end						1	0	1	0	1	0
						.end					

Table 7.15: Resulting G Function for *MVGUD* run on Benchmark *psu_rnd3*

.type mv .i 4 .o 1 .ilb a1 a5 a6 s2.2 .ob o0 .imv 2 2 2 5 .omv 2 .p 51					.type mv .i 4 .o 1 .ilb a4 a6 a7 s2.0 .ob o0 .imv 2 2 2 5 .omv 2 .p 51				
0	0	1	0,1,2	1	1	1	1	3	1
0	0	1	1,2	1	0	0	1	0	0
1	1	1	0	1	1	0	0	2	1
0	0	0	1,3	1	1	1	1	1	0
1	0	1	4	1	1	0	1	0	0
1	0	0	1,3	1	0	0	1	2	1
1	1	0	4	1	0	1	1	0	0
1	1	0	0	1	0	1	0	4	0
1	0	1	1	1	1	1	1	2	1
0	1	0	0	0	1	1	0	1	1
1	0	0	0,1	1	1	1	0	4	0
0	1	0	0,2,4	0	0	0	0	1	1
1	1	1	3	0	0	1	0	1	0
0	0	1	2	1	1	0	0	1	1
1	0	1	2	1	0	0	0	4	1
1	1	1	4	0	0	1	1	2	1
1	1	0	2	0	0	1	1	1	1
1	1	0	3	1	1	0	0	4	0
1	1	0	1,2	0	0	0	1	3	0
0	1	1	2	1	0	0	0	2	0
1	0	0	4	0	0	0	0	3	1
0	1	0	1,3	1	1	0	1	3	1
0	1	1	3	1	1	1	1	0	0
0	1	0	4	0	0	1	0	2	1
1	0	1	0	0	1	0	1	1	1
0	0	0	3	1	1	0	0	0	0
0	0	0	4	0	0	1	1	3	0
1	0	0	1	1	1	1	0	3	0
0	0	1	0,2	1	0	0	1	1	0
0	1	1	1,3	1	1	0	0	3	0
0	0	1	3,4	0	1	1	0	0	1
1	0	0	2	0	1	0	1	2	1
0	1	1	0	0	0	1	0	0	0
1	1	0	1	0	1	1	1	4	0
1	1	1	0,2	1	1	0	1	4	1
0	1	0	1	1	0	0	0	0	0
0	0	0	0	0	0	1	0	3	0
0	0	1	0,1	1	.end				
1	0	0	3	1					
0	1	1	0,4	0					
0	1	1	4	0					
1	1	1	0,1,2	1					
0	1	0	0,2	0					
0	1	0	2	0					
0	0	1	0	1					
0	1	0	3	1					
0	0	1	4	0					
1	0	1	3	0					
1	0	0	0	1					
0	0	0	0,2,4	0					
0	0	1	3	0					
.end									

Table 7.16: Resulting H Function for *MVGUD* run on Benchmark *psu_rnd3*

7.4 Summary and some Conclusions based on the results obtained in Chapter 7

The results obtained in this Chapter have provided some deeper insights into the decomposition process. The first very important conclusion is that Multi-Colorings or Clique-Coverings will nearly always produce better decompositions for Machine Learning Functions. Also the Multi-Coloring has proved that Decomposition of Relations is a very powerful concept, and why the introduction of generalized don't cares can improve the quality of the decompositions achieved. Whenever there are generalized don't cares in the output of a Relation, the simplest circuit can be selected, and implemented. While in the case of a function, once the decomposition is done there is no further choice that can be made. Hence while decomposing a function, generalized don't cares must be introduced, and that is what the Multi-Coloring is doing.

From Tables 7.5, 7.6 7.7 and 7.8 it was seen that as the number of variables in the Bound Set increases the number of generalized don't cares introduced in the decomposed blocks of the function increases. This can be seen from Figure 7.5 and Figure 7.6. In Figures 7.5 and 7.6 the number of variables in the Bound Set are increased along the X-axis, and the Y-axis displays the average *NOFP* or *COSC*. In order to get the average *NOFP*, for a constant number of variables in the Bound Set, the *NOFP* was added and then divided by the total number of program runs. Average *COSC* was calculated in the same way. Figure 7.5 is for when *MVGUD* was run on the Machine Learning Benchmarks with 70% of don't-cares, and Figure 7.6 is for when *MVGUD* was run on the Machine Learning Benchmarks with 90% of don't-cares. From these graphs the following observations can be made:

Observation 7.1 *As the number of variables in the Bound Set increases for a constant number of don't cares,* NOFP *increases exponentially.*

Observation 7.2 *As* NOFP *increases,* COSC *increases.*

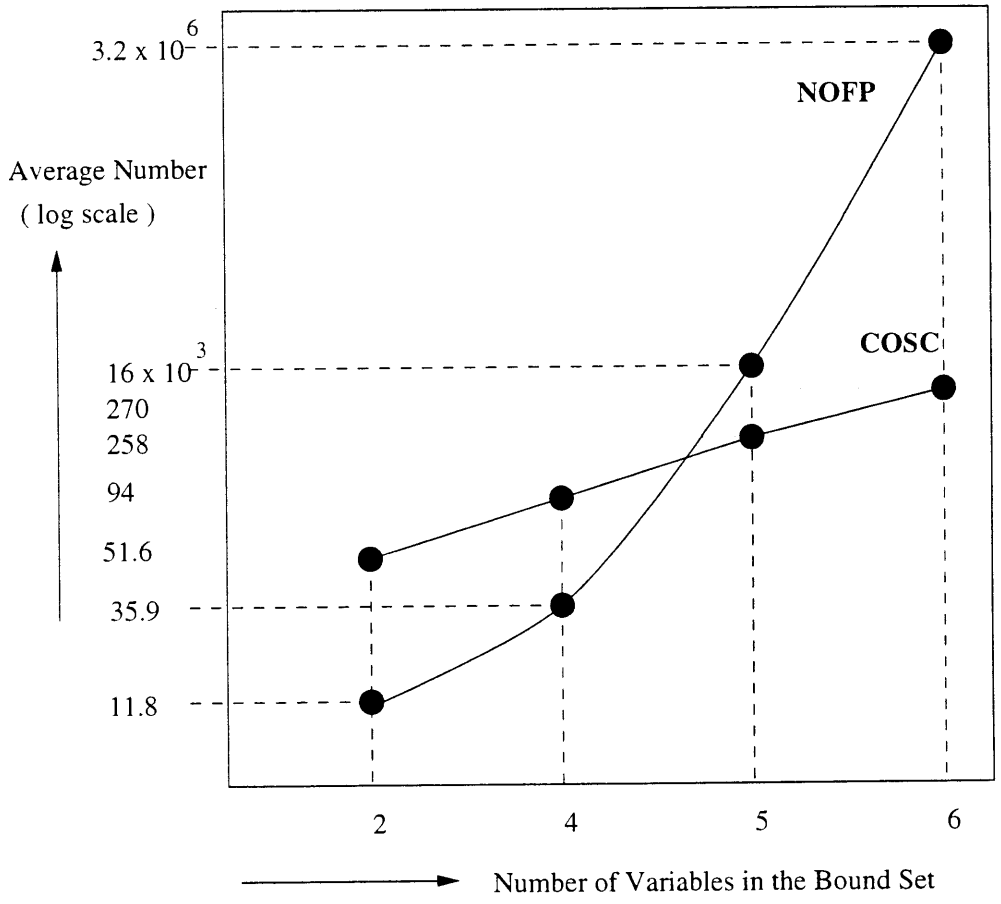


Figure 7.5: A Graph for 8 variable Machine Learning Benchmarks with 70% don't cares showing how *NOFP* and *COSC* vary with increasing number of variables in the Bound set

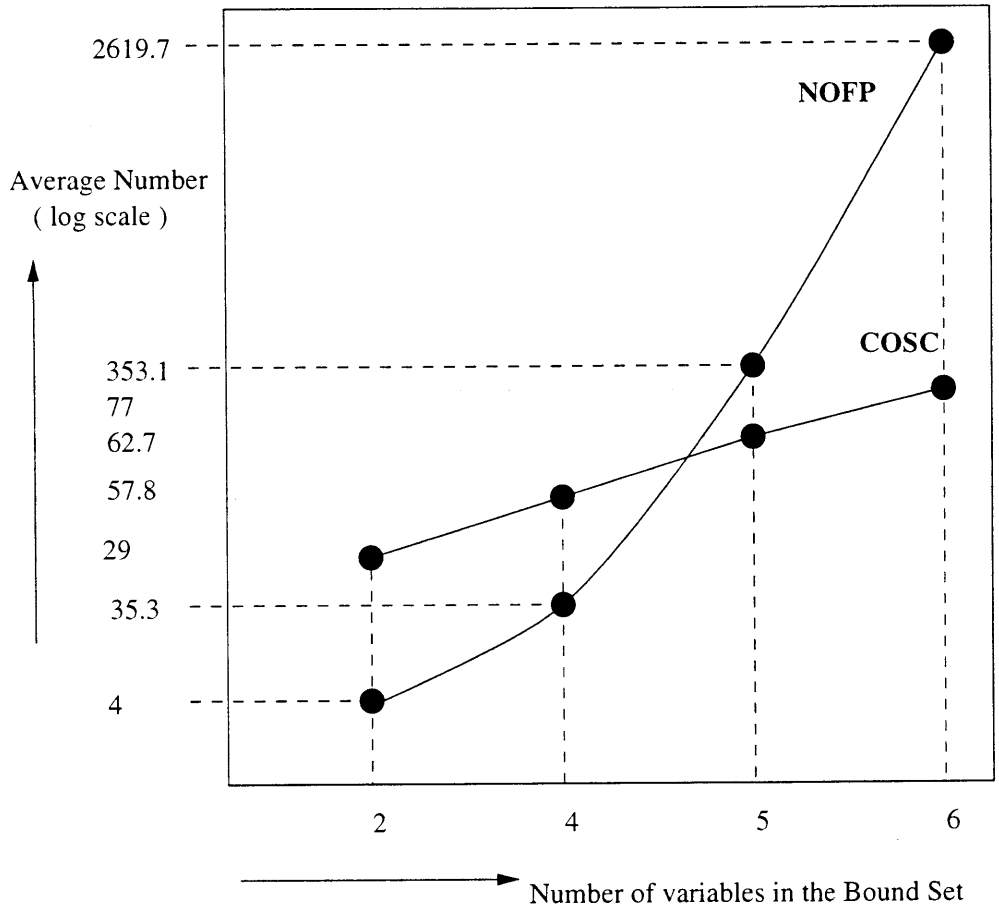


Figure 7.6: A Graph for 8 variable Machine Learning Benchmarks with 90% don't cares showing how *NOFP* and *COSC* vary with increasing number of variables in the Bound set

It was expected that as the number of don't-cares increase, *NOFP* should increase, but this was not so. The reason for this is that as the number of don't cares increase the size of the G and the H Tables is reducing, hence these reduced Tables may have a higher density of generalized don't cares than for the Tables with 70% don't cares, but due to the larger size of the Tables with 70% of don't cares they will have more generalized don't cares.

Observations 7.1 and 7.2 suggest that, in order to get better decompositions, it would be a good idea to have a new strategy for decomposition which starts from larger Bound Sets and then decomposes each G relation or function obtained. I think that this will lead to better decompositions. In order to try this new Strategy *MVGUD* will have to be modified. This would be one of the future works of the Functional Decomposition Group. The next Chapter concludes this thesis, and illustrates what I think the future work of the Functional Decomposition group should be.

CHAPTER 8

MY CONTRIBUTIONS TO THE FUNCTIONAL DECOMPOSITION GROUP, CONCLUSIONS AND FUTURE WORK

Since the Functional Decomposition Group was formed at Portland State University, we have spent a lot of time researching the different aspects of Functional Decomposition. Everybody in the group was involved in a different aspect of Functional Decomposition. My part was investigating the Column Multiplicity problem of Functional Decomposition. In order to investigate the Column Multiplicity problem properly, it was necessary to do it very systematically in a number of steps. As the first step I programmed the heuristic Dominance Coloring program and compared it with the heuristic Clique Partitioning. This testing showed us that a Clique Partitioning and a Graph Coloring both achieve nearly the same decompositions, and none of them proves to be better then the other. Since this did not solve the Column Multiplicity problem for us as the next step I wrote an Exact Graph Coloring. This provided us with a deeper insight into the Column Multiplicity problem. This testing proved to us that Exact Algorithms are not needed to find the Column Multiplicity for Curtis Decompositions. This was a very important result because it was always thought that an Exact Algorithm can significantly improve the decomposition. Even though this result provided us with a very useful idea, it also raised the very essential question that why is the Exact Algorithm unable to provide better decompositions? In order to find out the answer to this question, I tested *DOM*, *CLIP* and *EXOC* on the same graphs, generated during the process of Functional Decomposition in Chapter 5. This experiment told us that the graphs generated during the decomposition process

are much simpler than graphs generated randomly. This was another important conclusion, because it provided us with a deeper insight into the entire decomposition process, and not only to the part of finding the Column Multiplicity.

On showing that an Exact Graph Coloring is not necessary in Decomposition, the next question was whether a Multi-Coloring would improve the quality of decompositions with respect to coloring? In order to test the quality of the results obtained by using a Multi-Coloring, I developed two new cost functions. Then I developed and programmed a new algorithm for Multi-Coloring and tested it on MCNC and Machine Learning Benchmarks. Then I compared the heuristic Multi-Coloring with a heuristic Graph Coloring in decomposition of Circuit and Machine Learning Benchmarks. The two programs were evaluated by using the two new cost functions. The results of this experiment showed that a Multi-Coloring can improve the quality of decompositions achieved, for Machine Learning Functions. Thus I showed that in nearly all cases it is better to have a Multi-Coloring than having a Graph Coloring to find the Column Multiplicity in Decomposition of functions and relations. I do not know if this thesis has solved the Column Multiplicity problem in Functional Decomposition, but I think it has provided some very valuable insights into the decomposition problem, and it provides a direction that should be proceeded along in order to solve the decomposition problem better. The future direction that I propose that should be gone along to solve the column multiplicity problem better in the Decomposition process is to use a Multi-Coloring to find the Column Multiplicity and to start from larger bound sets, creating G relations, and then decomposing these G relations. This thesis has taught me that the decomposition process is a very powerful process, but it still has to attain its right status in the industry. But I think the decomposition process once solved better will become a very powerful tool in industry and research.

BIBLIOGRAPHY

[1] R.L. Ashenhurst, "The Decomposition of Switching functions," *Proc. Int'l Symp. Theory of Switching Functions,* pp.74-116,1959.

[2] H.A. Curtis, "A New Approach to the Design of Switching Circuits," *Princeton, N.J., Van Nostrand,* 1962.

[3] T. Luba, "New Approach to Boolean Function Decomposition for ROMs and PLAs," *Publishers of Institute of Telecommunication,* Warsaw Technical University, No.123, 1985.

[4] T.Luba, M. Mochocki, J. Rybnik, "Decomposition of Information Systems Using Decision Tables," *Bulletin of the Polish Academy of Sciences, Technical Sciences,* Vol. 41, No.3. 1993.

[5] W. Wan, M.A. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Functions Based on Graph-Coloring and Local Transformations and its Application to FPGA mapping," *Proc. of EURO-DAC'92,* Hamburg, pp. 230 - 235, September, 1992.

[6] W. Wan, Thesis, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Functions Based on Graph-Coloring and Local Transformations and its Application to FPGA mapping," Portland State University. 1992.

[7] M.A. Perkowski, T. Ross, D. Gadd, J.A. Goldman, and N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery," *Proc. Reed-Muller'95*, pp. 102-109, 1995.

[8] J.P. Roth, "Minimization over Boolean Trees," *IBM Journal.*4, 5, pp. 543-555, 1960.

[9] R.M. Karp, F.E. McFarlin, J.P. Roth, J.R. Wilts, "A Computer Program for the Synthesis of Combinational Switching Circuits," *In Proc AIEE Annual Symposium on Switching Circuits Theory,* pp. 182-194, 1961.

[10] J.P. Roth, and E.G. Wagner, "Algebraic Topological Methods for the Synthesis of Switching Systems. Part III: Minimization of Non Singular Boolean Trees," *IBM J. Res. Develop.* Vol 3, Oct 1962.

[11] R.M. Karp, "Functional Decomposition and Switching Circuit Design," *J. Soc. Industr. Appl. Math.,* Vol 11, No 2, pp. 291-335, June 1963.

[12] T. Sasao, "Functional Decomposition of PLAs," *Proc of the Intern. Workshop on Logic Synthesis,* Research Triangle Park, North Carolina, May 12-15, pp. 1987.

[13] M.A. Perkowski, H. Uong, "Generalized Decomposition of Incompletely Specified Multioutput, Multi-Valued Boolean Functions," *Unpublished Manuscript, Department of Electrical Engineering,* PSU 1987.

[14] M. Ciesielski, S. Yang, "A Generalized PLA Decomposition with Programmable Encoders," In the *Proc. of the Intern. Workshop on Logic Synthesis,* pp.1-13, May 1989.

[15] N.Sherwani, "Algorithms for VLSI Physical Design Automation".

[16] M. Axtell, T. Ross, "Pattern Theory in Algorithm Design", *National Aerospace and Electronics Conf (NAECON),* pp. May 1993.

[17] Y.T. Lai, M. Pedram, S. Sastry, "BDD-based Decomposition of Logic Functions With Application to FPGA Synthesis," *Proc of 30th DAC,* pp. 642-647, 1993.

[18] M.A. Perkowski, "A New Representation of Strongly Unspecified Switching Functions and it Application to Multi-Level AND/OR/EXOR Synthesis." *Proc. Reed Muller'95 Workshop,* Chiba, Japan, August 1995, pp. 143-151.

[19] S. Baase, "Computer Algorithms, Introduction to Design and Analysis", Second Edition, pp 351-352.

[20] J.C. Muzio, T.C. Wesselkamper, "Multiple-Valued Switching Theory," *Adam Hilger,* Boston, MA, 1986.

[21] P. Sapiecha, M. Perkowski, and T. Luba, "Decomposition of Information Systems Based on Graph Coloring Heuristics," *Symposium on Modeling, Analysis and Simulation*, CESA'96 IMACS Multiconference. Lille - France, July 9-12.

[22] S. Grygiel, M. Perkowski, M. Marek-Sadowska, T. Luba, L. Jozwiak, "Cube Diagram Bundles: A New Representation of Strongly Unspecified Multiple-Valued Functions and Relations," *ISMVL'97.*

[23] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, M. Nowicka, P. Burkey, R. Malvi, Z. Wang, J. Zhang, and C. Stanley, "Fundamental Operations on Multiple-Valued Cube Diagram Bundles," *Unpublished paper*, Portland State University, 1997.

[24] M. Perkowski, M. Marek-Sadowska, T. Luba, S. Grygiel, P. Burkey, R. Malvi, Z. Wang, J. Zhang, and C. Stanley, "GUD-MV:Multi Level Decomposition of Multi-Valued Functions and Relations: Part I: Cube Diagram Bundles," *Unpublished paper*, Portland State University, 1997.

[25] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, M. Nowicka, R. Malvi, Z. Wang, J. Zhang, "Decomposition of Multiple- Valued Relations" *ISMVL'97.*

[26] M. Perkowski, T. Luba, S. Grygiel, P. Burkey, M. Burns, N. Iliev, M. Kolsteran, R. Lisanke, R. Malvi, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. Zhang, "Unified Approach to Functional Decomposition of Switching Functions," Technical Report, Portland State University, 1995.